Boost
程序库探秘

深度解析C++准标准库

罗剑锋 著

C++98之实力，C++11之魅力

清华大学出版社

Boost 程序库探秘

——深度解析 C++准标准库

罗剑锋　著

清华大学出版社
北　京

内 容 简 介

Boost 程序库由 C++标准委员会部分成员所设立的 Boost 社区开发并维护，它功能强大、构造精巧、跨平台、开源并且完全免费，被称为“C++ ‘准’ 标准库”，已被广泛应用在实际软件开发中。

C++的最新标准（C++11）已经正式公布，而早在这之前，Boost 就已经使用库的形式实现了大部分新功能——而且是完全基于 C++98 标准实现的，内容涵盖智能指针、文本处理、并发、模板元等许多领域，其范围之广内涵之深甚至要超过 C++11 标准，极大地增强了 C++的功能和表现力。

本书基于 Boost1.47 版，深入探讨了其中的许多重要组件，包括迭代器、函数对象、容器、流处理、序列化以及 C++语言中最复杂最具威力的模板元编程，并专辟一章详细阐述 Boost 的开发实例，具有较强的实用性，可帮助读者更好更快地理解掌握 Boost 的高级用法。

全书内容丰富、组织得当、概念清晰、讲解细致，是广大 C++程序员和爱好者的必备好书。

图书在版编目（CIP）数据

Boost 程序库探秘——深度解析 C++准标准库 / 罗剑锋著. —北京：清华大学出版社，2012.3
ISBN 978-7-302-27485-8

Ⅰ. ①B…　Ⅱ. ①罗…　Ⅲ. ①C 语言－程序设计　Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2011）第 255580 号

责任编辑： 袁金敏
责任校对： 徐俊伟
责任印制：
出版发行： 清华大学出版社
　网　　址： http://www.tup.com.cn, http://www.wqbook.com
　地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
　社 总 机： 010-62770175　　**邮　　购：** 010-62786544
　投稿与读者服务： 010-62795954，jsjjc@tup.tsinghua.edu.cn
　质量反馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销： 全国新华书店
开　　本： 185×235　**印　张：** 38.5　　**字　数：** 937 千字
版　　次： 2012 年 3 月第 1 版　　**印　次：** 2012 年 3 月第 1 次印刷
印　　数：
定　　价：　　元

产品编号：044866-01

前言

缘起

2010年作者依据在实际开发中的经验编写了《Boost程序库完全开发指南——深入C++“准”标准库》（以下简称《指南》）一书，想为国内的C++程序员了解C++的最新技术进展尽一份自己的力量。图书付梓之后获得了许多的好评，同时也得到了很多有价值的批评意见——无论读者如何评价该书，对作者而言都是鼓励。

由于《指南》成书时间较仓促，Boost程序库又太过庞大，许多原本预想的内容限于篇幅不得不忍痛割爱予以移除，未能展现在读者面前，颇觉遗憾。于是时隔一年有余，作者将《指南》书中部分移除的内容补充完善，又增加了对Boost的新的研究心得，再次奉献给读者，希望同《指南》一样能够给国内C++程序员和爱好者带来一点帮助。

不比《指南》的一帆风顺，这次的写作过程异常艰辛，断断续续持续了一年半多，因为工作紧张、项目实施出差等原因数次暂停（累计差不多有数月的时间），中途再度拾笔写作时经常面临思路中断的烦恼。另一方面，书中的几个Boost组件比较复杂，对其的研究也是颇有难度，而把它表述成文字就更加困难，有时甚至有“茶壶煮饺子”的感觉。好在自己还算是一个有毅力的人，零敲碎打、慢慢琢磨，把零碎的时间积累起来总算完成了这部作品，个中甘苦一言难尽。

书名定为“探秘”，其意图在于不只是单纯地学习了解Boost库的用法，而是要更深入地探讨其内部的实现机制和原理，钻研其中的秘密所在。因此本书较之前作内容取舍和风格上略微有了些变化，不再是那些简单易用的组件，而是更偏重于C++深层次概念、高级工具、编译期的泛型编程和模板元编程，学习难度较《指南》有所提升，请读者阅读时留意。

拥抱 C++11

本书完成之时 C++最新标准已经公布，C++0x 被正式定名为 C++11，增加了许多新的特性，例如 nullptr、auto、decltype、attribute、constexpr、noexcept、static_assert、unique_ptr、move 语义、新式 for 循环、委托构造、增强的字符串语法、标准线程库等等。这些新特性对语言的改变有小有大，但总的来说是让 C++变得更友好更容易使用了，新手和专家都可以从中找到自己喜欢的东西。

虽然 C++11 大大增强了 C++的能力，但不可否认它同时也增加了语言的复杂度、学习成本以及使用成本，而且目前编译器和开发工具的支持也还不完善，个人认为在短期内（一到两年）还无法在国内的生产环境中大范围普及。今后几年里 C++11 最可能的情况是与 C++98 并存，C++群体逐渐了解熟悉新的语言特性和新的标准库，缓慢且平滑地由老标准过渡到新标准，而在这个过程中 Boost 无疑会扮演一个相当重要的角色。

作为 C++的“准”标准库，这些年 Boost 在 C++开发中的地位已经变得越来越重要，业已被广泛应用在实际产品中（用户包括 Adobe、Google、McAfee、SAP 等大公司），众多高质量的组件不仅极大地扩展了 C++98 在各个领域的能力，更为 C++98 添加了许多只有在 C++11 中才有的新特性，让我们的开发工作更加轻松、编写的代码更加优雅。Boost 也积极地参与到了 C++11 标准的实现中，许多库和概念早已经被接纳为 C++11 的一部分，比如 bind、function、iterators、random、smart_ptr、unordered 等，而其他的库也都为 C++11 的到来做好了准备，可以立即应用到支持新标准的编译器。从这个意义上来讲，Boost 是一个从 C++98 到 C++11 的最佳“引路人”，可以让我们以很低的成本无缝地过渡到 C++11。

C++和 Boost 博大精深，虽然已经有十余年的实践经验，但作者仍有“高山仰止”之感。由于篇幅和精力所限，Boost 中还有几个重量级库未能探究，包括语法解析器 spirit、元状态机 msm、函数式编程 phoenix、预处理元编程 preprocessor 和预处理器 wave，希望今后的某个时候能够有机会把这些库结合最新的 C++11 标准展示给读者。

致谢

首先仍然要感谢整个 C++群体和 Boost 社区，感谢他们长久以来的坚持和努力，让 C++和 Boost 变得越来越美好。

其次我要感谢我的家人：我的父母、弟弟、妻子，感谢你们对我写作工作的支持，你们永远是我生命中最重要的人；特别要感谢已经三岁的女儿，你是我生命中的小天使，愿你永远快乐，这本书是送给你的礼物。

最后感谢读者选择本书，还有那句“真心”的套话：限于作者水平有限，书中错漏在所难免，敬请批评指正。

罗剑锋

2011 年 11 月 2 日 于 北京 三里屯

目录

第 0 章

导读

0.1 关于本书

现在是 21 世纪的第二个十年，计算机编程语言领域已不复早期几家独大的局面，而是风起云涌、各领风骚，新的语言不断出现，同时也有老的语言逐渐衰落，但从一些权威统计机构的数据来看，三十年前诞生的 C++语言依然有着强大的生命力，稳稳保持着热门语言前三名的位置，即使是后来者 Java、C#、Python、Ruby 等也未能撼动它的王者地位。

C++能够获得这样的成就绝非运气，而是在于它自身的优异品质。它兼容“中级语言”C，具有良好的结构和绝佳的运行效率，可以开发系统级软件；它又开创了许多现代编程语言的范式，支持面向对象、泛型等技术，灵活方便，可以开发各种大型复杂的应用软件。在众多的编程语言中 C++可称得上是“全能选手”，可上可下，小至嵌入式系统，大至企业级应用，几乎没有什么事情是 C++做不到的。

C++的上一个国际标准诞生于 1998 年[①]，时至今日已经公布的最新标准 C++11 不仅将兼容 98 标准，更会为 C++带来更多的新特性和更强大易用的功能，例如增强的 Unicode 支持、统一的初始化语法、新的 auto/for/decltype 关键字、内建的 lambda 表达式、可变模板参数列表等[②]，但早在 C++11 推出之前，有着“C++ ‘准’标准库”美誉的 Boost 程序库就已经基本实现了这些功能——而且是完全基于旧标准使用库的形式实现的。

Boost 程序库充分利用了 C++的自扩展性这个最“神奇”的特性，在基本语言完全不变

---

① Java、C#、Python、Ruby、Lua 等语言并不是国际标准，只是公司标准、行业标准或事实标准。

② C++11 标准在正式公布前曾经被长期非正式称为 C++0x，其中的 x 表示年份不确定。

的情况下深入挖掘了语言的潜力，把泛型编程发挥到了极致，开发出了上百个功能强大的库，涉及内存管理、文本处理、容器与数据结构、图像处理、文件系统、并发、模板元编程等许多领域，范围之广内涵之深甚至要超过 C++11 标准。

随着 C++11 脚步的临近，在国外 Boost 早已经是大行其道[①]，而在国内 C++开发社区中 Boost 也逐渐流行起来。以作者个人所知，国内一些软件公司都或多或少地应用了 Boost 库的组件，也将能否掌握 Boost 作为评判个人能力的一个因素，但因为 Boost 库的博大精深远非一般的开源库可比，很多程序员也只能使用其中的少量简单组件，不能完全发挥 Boost 的真正实力，更有为数不少的人出于偏见仍然把 Boost 视作畏途[②]。

笔者 2010 年中编写了一本《Boost 程序库完全开发指南——深入 C++“准”标准库》（即推荐书目[1]，以下简称《指南》），偏重于对 Boost 的介绍和应用，基本不涉及实现，而本书作为该《指南》的延续则偏重于深入探究 C++语言和 Boost 的实现细节和原理，期冀达到“知其然更知其所以然”的境界，希望读者借助本书能够从 Boost 库中汲取更多有用的知识，提升自己的能力。

0.2 读者对象

本书定位于中高级读者，假设您已经对 C++的语言特性有较深层次的理解，并且具有了一定的 STL 和 Boost 知识。

在 Boost 程序库中面向对象的编程范式已经不是主要技术，更多的是使用泛型编程，所以本书的读者除了熟悉基本的面向对象技术外还应该对模板和泛型、模板的特化/偏特化、静态多态等 C++高级技术有所了解。

STL 是第一个真正把泛型编程技术表现的淋漓尽致的作品，现代的很多 C++库都深受其设计思想和结构的影响，Boost 当然也不例外，所以熟悉 STL 将非常有利于 Boost 程序库的学习。作为一个当代的 C++程序员，应该对 STL 的容器、迭代器和算法这三个最重要的部分捻熟于心（推荐书目[3]对标准库有详尽的介绍，读者可参考）。

Boost 程序库的很多组件笔者已经在《指南》中做了较详细的阐述，如 typeof、

① 例如著名的 NoSql 数据库 mongodb 就大量使用了 Boost 库的组件，包括 date_time、smart_ptr、static_assert、any、tuple、program_options、filesystem、function、bind、spirit、thread、utility 等。

② Boost 库目前的情形与十年前的 STL 非常相似：当年 STL 甫出，国外的程序员欢欣鼓舞，而国内的程序员却是担心效率、开发风格等诸多问题，畏手畏脚。时至今日，STL 已经成为了 C++程序员的基本素质，曾经的责难都已经烟消云散，相信假以不长的时日 Boost 也会获得广大程序员的认同。

foreach、shared_ptr、assign 等，本书中将会直接使用这些组件而少做或不做解释。如果读者对 Boost 所知不多，建议先阅读《指南》，然后再学习本书。

0.3 本书的风格

本书中所称的“标准”、“C++标准”指的是 C++98 标准，而不是已经出台的 C++11 标准。由于标准库已经成为了 C++的基础设施，故全书大部分代码均省略了标准库头文件和相应的“using namespace std;”语句，这一点请读者阅读时留意。不过个别情况下为了特别强调偶尔会加上名字空间前缀，如 std::copy()。

代码风格上本书遵循 C++标准库和 Boost 的惯例，自定义类和函数均采用小写形式，模板参数列表统一使用更规范的 typename 而不是 class，递增操作使用效率更高的前置式（++i）。因为目前大多数 C++编译器都不支持新的关键字 export，故本书中的模板类、模板函数的声明和实现均放在一起，而不是分离[①]。

本书中部分章节使用了 UML 类图来描述类的继承体系，绘图使用的工具是 StarUML（参见附录：程序员的工具箱），由于作者并不擅长美学，所以类图制作的不是太美观，仅能达到示意的程度，请读者见谅。

为了改善代码的可读性，本书中的示例代码版式有了一点小改进：某些需要强调的地方会用**粗体**或者*斜体*标明，读者可藉此迅速领会代码中的要点。

0.4 本书的开发环境

虽然目前已经推出了很多支持 C++11 部分特性的编译器，例如 VC10（Visual Studio 2010）、GCC4，但因为各种各样的原因（软件价格、学习成本等），并不是每个人都能够及时跟进编译器厂商的脚步，因此本书并没有使用最新的 C++编译器[②]。

本书作者的主要开发环境是 Windows XP + Visual Studio 2005（VC8.0），使用的标准库为 STLport5.21；另一个开发环境是 Mac OS X snow leopard（Darwin 10.8.0）

① 顺便一提，个人认为 export 这个关键字的引入相当糟糕，它违背了 C++的基本设计原则——“尽量不破坏已有的 C++代码”，因为现实代码中会有大量的函数或者变量被命名为 export，而现在，这些代码都因为名字冲突而无法使用，如果它像 using 关键字一样改用“exporting”可能会更好。另有较确切的消息称，C++11 标准中可能会将这个关键字废除。

② 不要忘记下面的事实：甚至现在还有大量的开发人员在使用十多年前的“古董”VC6。

+ Xcode4（gcc4.2.1），使用自带的标准库实现，另有少量代码在 Linux 2.6.18 x86_64（gcc4.1.2）下编译通过。Boost 程序库则基于官方于 2011 年 7 月发布的 Boost1.47 版。

Boost 库中的大部分组件都是以头文件的方式实现的，不需要编译，直接包含头文件即可。少数库需要编译才能使用，Boost 提供类似 make 的 bjam 工具来完成库的编译和安装，可以从 Boost 的网站上下载到适用于自己操作系统的 bjam 程序自行编译，但本书不使用 bjam 编译 Boost 库，而是把所有实现 cpp 合并到一个 cpp 中集中编译，这种方式被称为 unity build（也被作者形象地称为“集结号”式编译）[①]。

0.5 本书的结构

本书假设读者已经具有了一定的 STL 和 Boost 知识，所以不再对 Boost 库的安装、编译和环境设置做介绍，而是直接切入主题讲解库的使用。

全书共分 14 章，每章可能包含若干 Boost 个组件，章节的最后是对本章的总结。组件的通常讲解顺序如下：先简要介绍其功能，然后说明其头文件和编译方法（如果需要编译的话），列出类的摘要（可能还配以 UML 图解），再使用例子讲解用法和注意事项。

本书大致可以分为以下八个部分。

■ 第一部分，第 0 章:介绍本书的基本内容和一些注意事项

这个部分就是读者正在阅读的这章，您很快就会看完（笑）。

■ 第二部分，第 2～4 章：介绍 Boost 中的一些实用工具

本部分从 C++的深层次概念入手，介绍 Boost 库提供的各种实用工具，涉及类型转换、对象的创建/初始化/删除、指针相关工具、迭代器的概念和构造、函数对象等内容，学习它们可以对 C++的底层语言细节有更深刻的了解，有助于构建更稳固健壮的程序。

■ 第三部分，第 5～7 章：介绍 Boost 中的新式容器

本部分较详细地阐述了《指南》未涉足的三类 Boost 新式容器：指针容器、侵入式容器和多索引容器，它们从不同的方向扩展了标准容器，功能更强大内容更庞杂，用起来有许多额外的考虑，读者会发现它们能够应用在许多特殊的场景。

① unity build 不仅适用于 Boost 库编译，也可以把它应用在实际开发工作中，它对于组织大型的 C++项目特别有用，要求 C++代码必须写的相当规范。

■ 第四部分，第 8～9 章：介绍 Boost 中的输入输出功能

本部分虽然只有两章，但其分量绝对不轻，论述了 Boost 库提供的流处理和序列化两大功能。流处理是 C++诞生之初就有的功能，但多年来一直未被重视，boost.iostreams 在标准库的基础上构造了功能完备富有弹性的流式处理框架。序列化是 C++社区长期追求的目标之一，boost.serialization 实现了完整可用的任意类型的序列化支持，可用于数据的持久化。

■ 第五部分，第 1 章、第 10 章和第 11 章：介绍泛型编程和模板元编程

本部分介绍 Boost 中最高级的泛型编程和模板元编程，包括 type_traits、concept_check 和 mpl 等库，需要读者对 C++的泛型技术具有较深刻的认识。模板元编程虽然已经出现了很多年，但在国内目前仍然算是一个较新的领域，相关的实践经验也不是很多，作者在这里也仅仅是做一个较为粗略的介绍。

■ 第六部分，第 12 章：实际使用 Boost 开发

本部分综合运用了多个 Boost 组件，如 smart_ptr、pool、factory、unordered、bind、function、thread、asio，通过开发两个 TCP 服务器程序示范了如何把 Boost 应用于真正的编程开发工作中，算是一个小小的 boost cook book。

■ 第七部分，第 13 章：介绍作者的一些开发经验

本部分仿造推荐书目[6]和[8]，基于作者多年使用 Boost 的经历以条款的形式给出了一些使用 STL 和 Boost 的经验，不一定完全正确，权做抛砖引玉之举，希望能够给读者起到一点借鉴的作用。

■ 第八部分，附录

书末的附录首先收录了作者认为值得阅读的 C++经典作品，然后是 Boost1.47 版全部组件的索引，最后是作者对软件开发常用工具软件的一个简单推荐。

0.6 如何阅读本书

对于所有读者来说，作者推荐首先阅读第 0 章和第 1 章，特别是第 1 章——虽然它属于第五部分，而且内容较深，但由于模板元编程在本书中的重要地位而被提到了全书的最开始章节，随后的许多章节都会用到其中的概念和工具，熟悉模板元编程的基本概念对于理解本书是非常有帮助的。

接下来读者可以随个人意愿，或者按照书籍的物理顺序循序渐进逐页阅读，或者查阅目录，然后跳到感兴趣的章节。书中包含了大量作为示例的 C++源代码，都附加了详细的注释，希望读者能够耐心仔细阅读。另外本书还精心编制了交叉索引，阅读时涉及其他章节的内容都会指明具体位置和页码，便于读者快速参考。

由于 Boost 库的很多组件的原理、用法较复杂，读者可结合附录的推荐书目阅读，最好再打开自己最熟悉的编辑器或开发环境，直接查看库源代码以加深理解。

好，深吸一口气，welcome back，欢迎再次来到 Boost 的世界!!!

第1章

模板元编程（I）

C++是一种功能强大到几乎可称为“万能”的语言，支持许多种不同的编程范式，而由泛型编程衍生出的模板元编程（template meta-programing，简称元编程）则无疑是其中最复杂、最强大和最具威力的一种，可算得上是C++的“终极”技巧（传说中的“大杀器”，笑）。

所谓“元程序”——metaprogram，意思就是“a program about a program”，它有着完全不同于普通程序的许多特点，是一种全新的编程体验。作为本书的第一章，首先来讨论模板元编程的一些基本概念，它们是许多 Boost 程序库组件的基础，了解模板元编程有助于我们对 Boost 的进一步学习。

1.1 模板元编程概述

元编程也可译为“超程序”、“超编程”或“产生式编程”，这个译法一定程度上反映了其本质——它是一种位于普通程序之上、超越普通程序的程序，是一种可以操纵、产生程序的程序。

C++中的模板元编程是无意中被“发现”而不是“发明”出来的，是一种对“类型”计算的程序，关于它的诞生有许多有趣的传奇和故事。模板元编程本质上是泛型编程的一个子集，从广义上来说，所有使用 template 的泛型代码都可以称作是元程序，因为泛型代码不是真正可编译执行的代码，它们只是定义了代码的产生规则，是用来生成代码的“模板”。

然而模板元编程又不完全等同于泛型编程，它是一种“函数式编程”，并且已经被证明是图灵完备的，也就是说它具有足够的“计算”能力，可以“计算”任何东西。模板元编程的执行是在编译期，它把编译器变成了元程序的解释器，可以把 C++的类型体系像面团一样

捏来捏去，肆意打碎再任意组合起来，拥有近乎不可思议的“魔力”。

虽然模板元编程使用的是标准的 C++语言，遵循同样的语法规则，但它在编程思想、编程范式等很多方面都与普通的运行时 C++程序有很大的不同。模板元编程产生的元程序是在编译期执行的程序，操作的对象也不是普通的变量，因此不能使用运行时的 C++关键字（如 if、else、for），可用的语法元素相当有限，最常用的是：

- enum、static，用来定义编译期的整数常量；
- typedef，最重要的元编程关键字，用于定义元数据；
- template，模板元编程的“起点”，主要用于定义元函数；
- “::”，域运算符，用于解析类型作用域获取计算结果（元数据）。

下面简单介绍模板元编程领域中的三个最基本的概念，熟悉它们才能够理解模板元编程和元程序。当然元编程中远不止这些，还有 lambda 表达式、容器、迭代器、算法、视图等更高级的概念，它们将在本书的第 11 章叙述。

1.1.1 元数据

元编程可操作的数据就称为“元数据”（meta data），也就是 C++编译器在编译期可操作的数据，它是元编程的基础。

元数据都是不可变的，不能够就地修改，最常见的元数据是整数和 C++的类型（type）。

对于整数大家都很熟悉，普通程序在运行时也可以很容易地处理，但在模板元编程中的元数据更多的是以类型（type）的面目出现。这些元数据不是普通的运行时变量，而是如 int、double、class（非模板类）这样的抽象数据类型——这是模板元编程与普通运行时编程的一个最根本的不同点，也是元编程的威力所在（类型的计算）和令初学者感到最困惑的地方。

如果对元数据再进行细分归类，则元数据又可以分成整数元数据、值型元数据（int、double 等 POD 值类型）、函数元数据（函数类型）、类元数据（class、struct 等用户自定义类型）等等。为了更明确地表述元数据的概念，本书后面的“元数据”一词特指非整数的类型元数据。

使用 typedef 关键字可以任意定义（声明）元数据，很像运行时的变量定义语句，例如：

```
//元数据 meta_data1，值为 int
typedef int meta_data1;
//元数据 meta_data2，值为 vector<float>
typedef std::vector<float> meta_data2;
```

1.1.2 元函数

元函数（meta function）是模板元编程中用于操作处理元数据的“构件”，可以在编译期被“调用”，因为其功能和形式类似运行时的函数而得名，是元编程中最重要的构件。

元函数实际上表现为 C++中的一个类或者模板类，它的通常形式是：

```
template<typename arg1, typename arg2,...>   //元函数参数列表
struct meta_function                          //元函数名
{
  typedef some-define type;                   //元函数返回的元数据
  static int const value = some-int;          //元函数返回的整数
};                                            //使用分号结束元函数的定义
```

编写元函数就像是编写一个普通的运行时函数，但形式上却是一个模板类：

函数参数列表圆括号()变成了模板列表声明的尖括号<>，函数的形参变成了模板参数（即元数据）。因为元编程不能使用运行时关键字，所以元函数不能像普通函数那样使用 return 返回计算结果[①]，而是需要在元函数（模板类）内部用 typedef 定义一个名为 type 的类型（元数据）或者名为 value 的值作为返回[②]。还要注意的一个小区别是元函数最后要以分号（;）结束，这是因为元函数实质上是一个类，C++的语法要求类定义需要分号。

通常来说，所有的元函数都有::type 返回值，但不一定有::value 返回值，但为了方便通常值元函数也会提供直接的::value，相当于::type::value。

进一步类比普通函数有助于我们更好地理解元函数：

元函数同样也有形参、实参的概念，元函数的形参就是模板参数列表中的模板参数，实参就是元函数被调用时的真正类型（元数据），元函数的调用就是编译器对模板的实例化计算依赖的类型。元函数也可以没有返回值（即不定义内部类型 type），也可以有重载，也可以

---

① 实际上 return 也无法返回非整数的元数据。
② 这里的 type 和 value 名字仅仅是 Boost 模板元编程领域的一个非强制性的约定，统一遵循这个约定会方便元函数的使用，用户无须查看源代码就可以确定元函数内部必定有名为 type 或 value 的内部定义。当然我们也可以自己另行建立一套自己的约定，但这通常没有必要。

有缺省参数，也可以分为无参、单参、多参等类别，但元函数没有普通函数参数传值、传引用的区别，也没有函数指针的概念，而且如果需要，元函数可以使用 typedef 关键字返回任意多个返回值，并且这些值没有顺序关系（可以将只返回::type 的元函数称为标准元函数，而返回多个元数据的元函数称为非标准元函数）。

下面的代码是一个最平淡无奇的值元函数，它计算两个整数的和：

```
#include <boost/config.hpp>                    //提供 BOOST_STATIC_CONSTANT 宏
template<int N, int M>                         //两个整数元数据
struct meta_func1                              //元函数 meta_func1
{
  BOOST_STATIC_CONSTANT(int, value = N + M); //编译期计算整数之和
};
```

计算结果可以这样得到：

```
cout << meta_func1<10, 10>::value << endl;   //使用::value 获得计算结果
```

读者需注意元函数 meta_func1<>的执行过程，它的计算在编译的时候就已经完成了（即模板实例化），实际程序执行时没有计算动作而是直接使用结果，如果这是一个大型的元函数，那么在编译期节约的计算工作量就会相当可观，可以显著提高程序运行时的效率。

因为元函数的计算发生在编译期，所以下面的代码不能成立：

```
int i = 10, j = 10;                 //两个运行时变量
meta_func1<i, j>::value;            //错误，元函数无法处理运行时普通数据
```

下面的代码示范了另一个元函数，它返回元函数参数中的第一个元数据：

```
template<typename T1, typename T2>    //两个形参，T1 和 T2
struct select1st                      //元函数 select1st
{
  typedef T1 type;                    //返回 T1
};
```

请读者谨记一点，元函数就是一个形式上很像函数的一个模板类，它用于计算类型。在本书之后的叙述中，为了把元函数与普通的 C++类区分，元函数名称后面将总有模板参数列表符号<>，这种表述形式很像是一个函数。

1.1.3 元函数转发

这是模板元编程中一个经常用到的惯用法，相当于运行时的函数转发调用，但在模板元

编程中则要使用 public 派生实现，模板参数传递给父类完成元函数的调用，这样子类会自动获得父类的::type 定义，也就完成了元函数的返回。

例如，下面的代码把元数据调换位置后转发给之前定义的元函数 select1st<>，相当于 select2nd<>的功能：

```
template<typename T1, typename T2>
struct forward :       //元函数转发，使用 struct 的默认 public 继承
  select1st<T2, T1>    //参数位置变动
{};
```

元函数转发等价于如下的写法，但因为使用了类继承所以更加简洁易读：

```
template<typename T1, typename T2>
struct forward                                     //元函数，不使用转发
{
  typedef typename select1st<T2, T1>::type type;   //调用元函数计算
};
```

本书后面的章节中将会有元函数转发的多处应用示例，如 1.2.8 小节。

1.1.4 方便的工具

模板元编程是一种全新的 C++编程范式，但却仍然使用原有的 C++语法，通篇的 typedef、template 关键字使元程序看起来有如“天书”，对于初学者来说通常很难在短时间内适应这种转变。作者在实际开发工作中使用宏定义了一些模板元编程的“伪关键字”，一定程度上能够使模板元程序更加清晰易懂，更能够显示其元编程的本意，也确实收到了较好的效果。

这些自定义“伪关键字”均以 mp_开头（如果读者不喜欢这个前缀也可以改用 meta_或者其他自己觉得更有意义的单词），由于使用了小写的形式让“伪关键字”代码形式上更像关键字，代码看起来更加漂亮：

```
// mp_utils.hpp
#define mp_arglist        template        //元函数参数列表开始
#define mp_arg            typename        //元函数参数声明
#define mp_function       struct          //元函数定义
#define mp_data           typedef         //元数据定义

#define mp_return(T)      mp_data T type  //元函数返回
```

```
#define mp_exec(Func) Func::type        //获得元函数返回结果
#define mp_eval(Func) Func::value       //获得元函数返回值
```

这个头文件（mp_utils.hpp）分别把 template、typename、struct 和 typedef 这四个元编程中最常用的关键字进行了重命名：mp_arglist 表示元函数的参数列表开始，mp_arg 表示元函数的参数，mp_function 表示定义一个元函数，mp_data 表示定义一个元数据。后三个宏 mp_return、mp_exec 和 mp_eval 则定义了元编程中约定的返回值用法，较原写法更清楚。

使用这些“伪关键字”，之前的元数据和元函数可以改写成如下的形式：

```
mp_data int meta_data1;                 //元数据 meta_data1，值为 int

mp_arglist<mp_arg T1, mp_arg T2>        //元函数参数是 T1 和 T2
mp_function select1st                   //元函数 select1st
{
  mp_return(T1);                        //返回 T1
};
```

很明显，使用了元编程“伪关键字”后元程序看起来更清楚，很容易地就能够把它们与普通的泛型代码区分开来。

不过读者需要注意的是由于宏自身的限制，后三个“伪关键字”的作用有限，它们只能处理简单的参数，如果是带有逗号“,”的模板类就会失效。

本章之后的部分代码会把这些自定义的元编程“伪关键字”与 C++的标准关键字混用，读者阅读时需注意。

1.1.5 应用示例

本小节通过两个简单的例子来示范元编程的基本使用，仅作为演示，并不具备太多的实际价值，更多的元编程示例可见本书的后续章节。

编译期比较大小

下面首先来编写一个值元函数，它在编译期比较两个 int 型整数的大小，返回其中的较小者，代码非常简单：

```
mp_arglist<int L, int R>
mp_function static_min                          //元函数 static_min
```

```
{
  static const int value = (L < R) ? L : R;  //?:操作符可用于编译期
};
```

static_min<>用起来也很容易，例如：

```
assert((static_min<10, 20>::value == 10));
```

虽然 static_min<>的代码很简单，但如同标准库的 std::min()一样，这样小而有用的元函数对于元编程也非常重要的，Boost 专门在头文件<boost/integer/static_min_max.hpp>中提供了可以处理任意有符号整数和无符号整数的 min<>/max<>元函数，原理相同，但实现更加坚固可靠。

操作类型的元函数

接下来编写一个元函数 demo_func<>，如果输入的元数据 T 是指针类型则返回 const T，否则返回 const T*。因为元编程中不能使用 if-else 分支语句，所以我们的主要实现手段就是模板特化，不同的条件特化不同的实现代码：

```
mp_arglist<mp_arg T>      //单参元函数
mp_function demo_func     //元函数 demo_func
{
  mp_return(const T*);    //通常情况返回 const T*
};
mp_arglist<mp_arg T>
mp_function demo_func<T*> //对 T*情况进行模板特化
{
  mp_return(const T);     //返回 const T
};
```

对元函数 demo_func<>的验证可以使用 1.2.5 小节介绍的 is_same<>元函数，它用来比较两个元数据是否相等[1]：

```
assert((is_same<mp_exec(demo_func<int>), const int*>::value));
assert((is_same<mp_exec(demo_func<int*>), const int>::value));
```

这里的 assert 语句必须使用两对括号来包围断言，否则会因为宏无法识别模板语法，

① 示例代码中的 assert 宏完全可以使用 BOOST_STATIC_ASSERT 替代，而且会更明显地表明元函数编译期运行的意图，不使用 BOOST_STATIC_ASSERT 完全是照顾版面布局的原因（单词太长了）。

导致错误的逗号解析造成编译失败。

1.2　type_traits

了解了基本的模板元编程概念后我们来学习第一个模板元编程库：type_traits，它以库的方式实现了人们原本以为必须扩展 C++语言才能实现的类型特征提取功能，是泛型编程和模板元编程所必需的基础设施，已经被收入 C++11 tr1。

type_traits 位于名字空间 boost，为了使用 type_traits 组件，需要包含头文件 <boost/type_traits.hpp>，即：

```
#include <boost/type_traits.hpp>
using namespace boost;
```

1.2.1　概述

type_traits 提供一组特征（trait）类——即元函数，可以在编译期确定类型（元数据）是否具有某些特征，例如是否是原生数组，是否是整数。此外它还提供了判断类型之间的关系和操作类型的元函数，可以检查两个类型是否是同一个类型，或者为类型增添或移除 const、volatile 等修饰词。因为这些特征类都是元函数，所以它们都在编译期执行，不会有任何运行时的效率损失①。

type_traits 库提供数十个元函数，可以按照返回类型和功能分类。

根据返回类型 type_traits 库所提供的元函数可分为以下两大类：

- 检查元数据属性的值元函数：以::value 返回一个 bool 值或者一个整数；
- 操作元数据的标准元函数　：对元数据进行计算，以::type 返回一个新的元数据。

type_traits 库中以 is_和 has_开头的元函数均属于值元函数，其他则属于标准元函数，但少数元函数也有例外。

根据元函数实现的功能 type_traits 库所提供的元函数可分为以下六类：

- 检查元数据的类别　　　　：均以 is_开头，都是值元函数；

① Boost 自带文档中把 type_traits 库操作的对象称为 type（类型），但本书作者认为名字太过于平淡而不突出，因此依据模板元编程统称为元数据，有利于保持概念一致性，希望读者阅读时注意。

- 检查元数据的属性　　　　：大部分以 is_和 has_开头，都是值元函数；
- 检查元函数之间的关系　　：均以 is_开头，都是值元函数；
- 转换元数据　　　　　　　：都是标准元函数，返回转换后的类型；
- 用指定的对齐方式组合类型：包括 type_with_alignment<T>和 aligned_storage<T>两个元函数，本书不做介绍；
- 解析函数元数据　　　　　：是非标准元函数。

接下来的数个小节我们将以功能分类逐个介绍 type_traits 库里的元函数。

1.2.2 元数据类别（Ⅰ）

type_traits 库提供 16 个值元函数检查元数据 T 的基本类别（primary traits），被检查的元数据前可以用 const、volatile 关键字修饰，元函数用::value 返回 bool 类型的检查结果。

检查基本类型

检查基本 C++类型的元函数有以下四个：

- is_integral<T>：检查 T 是否是 bool、char、int、long 等整型，这些整型前可以有 signed、unsigned 等关键字；
- is_float<T>/is_floating_point<T>：检查 T 是否是 float、double、long double 等浮点型；
- is_void<T>　　：检查类型 T 是否为 void 类型。

示范这四个元函数用法的代码如下，其中的类型应视为元数据常量：

```
assert(mp_eval(is_integral<const char>));      //有 const 修饰
assert(mp_eval(is_integral<unsigned long>)); //有 unsigned 修饰
assert(!mp_eval(is_integral<int*>));          //指针类型不是整型

assert(mp_eval(is_float<double>));            //double 属于浮点类型
assert(mp_eval(is_floating_point<float>));    //float 属于浮点类型
assert(!mp_eval(is_floating_point<int>));     //整数类型不是浮点类型

assert(mp_eval(is_void<void>));               //检查 void 类型
```

```
assert(!mp_eval(is_void<void*>));            //void*指针类型不是 void
```

检查其他类型

接下来的十个元函数检查其他的 C++类型：

- is_array<T> ：检查 T 是否是一个原生数组（包括一维和多维数组）；
- is_class<T> ：检查 T 是否是一个 class 或者 struct；
- is_enum<T> ：检查 T 是否是一个枚举类型；
- is_union<T> ：检查 T 是否是一个联合类型[①]；
- is_complex<T> ：检查 T 是否是一个标准库的复数类型（std::complex<U>）；
- is_pointer<T> ：检查 T 是否是一个指针或函数指针类型，但不是成员指针；
- is_lvalue_reference<T>：检查 T 是否是一个左值引用类型；
- is_rvalue_reference<T>：检查 T 是否是一个右值引用类型；
- is_reference<T> ：检查 T 是否是一个引用类型（左引用或右引用）[②]；
- is_function<T> ：检查 T 是否是一个函数类型，但不是函数指针或引用。

下面的代码示范了其中部分元函数的用法：

```
assert(mp_eval(is_array<double[]>));                //一维数组类型
assert(mp_eval(is_array<int[2][3]>));               //多维数组类型

assert(mp_eval(is_class<struct dummy>));            //一个空类
assert(mp_eval(is_class<std::vector<int> >));       //标准容器类

assert(mp_eval(is_complex<std::complex<double> >)); //复数

assert(mp_eval(is_pointer<int*>));                  //整型指针
assert(mp_eval(is_pointer<int(*)(int)>));           //函数指针
```

---

① is_union<T>的功能实现需要编译器的支持，有的编译器会把 union 识别为 class/struct。
② type_traits 库原本只有 is_reference<T>，Boost1.45 版后增加了对左引用(T&)和右引用(T&&)的区分，但因为右引用语法是 C++11 里的新特性，故无法在 C++98 标准中进行测试，请读者见谅。

```
assert(mp_eval(is_reference<float&>));                    //引用
assert(mp_eval(is_reference<std::deque<int> const&>));    //容器常引用

assert(mp_eval(is_function<void(int,double)>));           //函数类型
```

检查成员指针类型

最后是两个专门用于识别类成员指针类型的元函数：

- is_member_object_pointer<T>   ：检查 T 是否是指向成员变量的指针；
- is_member_function_pointer<T>：检查 T 是否是一个成员函数指针。

示范这两个元函数用法的代码如下：

```
struct dummy        //一个简单的类
{
  int x;            //int 成员变量
  double y;         //double 成员变量
  void func(){}     //成员函数
};

assert(mp_eval(is_member_object_pointer<int(dummy::*)>));
assert(mp_eval(is_member_object_pointer<double(dummy::*)>));
assert(mp_eval(is_member_function_pointer<void(dummy::*)()>));
```

1.2.3 元数据类别（II）

在 16 个基本元函数之上，type_traits 库又提供了六个检查复合类别（composite traits）的元函数，它们相当于多个基本类别的组合，使用::value 返回 bool 类型的检查结果，列举如下：

- is_arithmetic<T>   ：检查 T 是否是算术类型，相当于 is_integral<T> || is_float<T>；
- is_fundamental<T> ：检查 T 是否是基本类型，相当于 is_arithmetic<T> || is_void<T>；
- is_compound<T>   ：检查 T 是否是复合类型，即非基本类型，相当于！is_fundamental<T>；
- is_member_pointer<T>：检查 T 是否是成员指针，包括指向数据成员和函数成员的指针，相当于 is_member_object_pointer

<T>|| is_member_function_pointer<T>;

- is_object<T> ：检查 T 是否是实体对象类型，即引用、void 和函数之外的所有类型，相当于!is_reference<T> && !is_void<T> && !is_function<T>;
- is_scalar<T> ：检查 T 是否是标量类型，即算术类型、枚举、指针和成员指针，相当于is_arithmetic<T> || is_enum<T> || is_pointer<T> || is_member_pointer<T>。

示范这些检查复合类别元函数的代码如下：

```
assert(mp_eval(is_arithmetic<char>) );            //char 是算术类型
assert(mp_eval(is_arithmetic<float volatile>));   //float 是算术类型

assert(!mp_eval(is_arithmetic<void const>));      //void 不是算术类型
assert(mp_eval(is_fundamental<void const>));      //void 是基本类型

assert(mp_eval(is_member_pointer<int(dummy::*)>)); //成员指针

assert(mp_eval(is_compound<std::string>));        //标准字符串是复合类型
assert(mp_eval(is_object<std::string>));          //标准字符串也是对象类型

assert(mp_eval(is_scalar<int>));                  //int 是标量类型
assert(!mp_eval(is_scalar<std::vector<int> >));   //标准容器不是标量类型
```

1.2.4 元数据属性

除了检查类别，type_traits 库里还有更多的元函数用于获取元数据更细致的属性。这些元函数都是值元函数，以 is_和 has_开头的元函数使用::value 返回 bool 类型的检查结果，其他元函数使用::value 返回整数。

检查数组的属性

- rank<T> ：如果 T 是数组，那么返回数组的维数，否则返回 0;
- extent<T,N>：如果 T 是数组，那么返回数组第 N 个维度（从 0 计数）的值，否则返回 0。

检查基本的修饰词

- is_const<T> ：检查 T 是否被 const 修饰；
- is_volatile<T>：检查 T 是否被 volatile 修饰；
- is_signed<T> ：检查 T 是否是有符号整数；
- is_unsigned<T>：检查 T 是否是无符号整数。

检查 class 的属性

- is_pod<T> ：检查 T 是否是一个 POD 类型[①]；
- is_empty<T> ：检查 T 是否是一个空类（可参见 2.1.1 小节）；
- is_abstract<T> ：检查 T 是否是一个抽象类（有纯虚函数）；
- is_polymorphic<T>：检查 T 是否是一个多态类（有虚函数）。

检查 class 的四大成员函数（构造、析构、拷贝构造和赋值）

- has_nothrow_constructor<T>：检查 T 是否有不抛出异常的构造函数；
- has_nothrow_default_constructor<T>：检查 T 是否有不抛出异常的缺省构造函数；
- has_trivial_constructor<T>：检查 T 是否有“平凡”的构造函数[②]；
- has_trivial_default_constructor<T>：检查 T 是否有“平凡”的缺省构造函数；
- has_trivial_destructor<T> ：检查 T 是否有“平凡”的析构函数；
- has_virtual_destructor<T> ：检查 T 是否有虚析构函数；
- has_nothrow_copy<T> ：检查 T 是否有不抛出异常的拷贝构造函数；

---

① POD 是术语 Plain Old Data 的缩写，但没有明确的定义。通常来说基本类型(is_fundamental<T>::value == true) 都是 POD，而复合类型的 POD 则没有构造函数、析构函数、虚函数，内存布局是连续的，与 C 语言定义的类型等价。

② “trivial”这个术语在 C++中是指“平凡”、“非特殊”的含义，“平凡”的构造、析构函数没有任何操作，“平凡”的拷贝构造函数、赋值函数类似于 memcpy 操作，因而可以被编译器优化。

- has_nothrow_copy_constructor<T>：同 has_nothrow_copy<T>;
- has_trivial_copy<T>：检查 T 是否有“平凡”的拷贝构造函数;
- has_trivial_copy_constructor<T>：同 has_trivial_copy<T>;
- has_nothrow_assign<T>：检查 T 是否有不抛出异常的赋值函数;
- has_trivial_assign<T>：检查 T 是否有“平凡”的赋值函数。

其他属性

- has_new_operator<T>：检查 T 是否重载了 operator new;
- is_stateless<T>：检查 T 是否是一个无状态类，即一个“平凡”的空类;
- alignment_of<T>：T 在内存中字节对齐的倍数。

以上一共列举了 25 个属性检查元函数，下面的代码简要示范了其中部分元函数的用法：

```
//获得数组 int[2][3]的维数
assert(mp_eval(rank<int[2][3]>) == 2);
//因 extent<>有多个元参数，mp_eval 失效，只能使用原始形式
assert((extent<int[2][3], 1>::value == 3));          //获取第二个维度

assert(!mp_eval(is_pod<std::string>));               //标准字符串不是 POD 类型
assert(mp_eval(is_empty<std::plus<int> >));          //函数对象是空类
assert(mp_eval(is_polymorphic<std::iostream>));      //标准流是多态的

//标准字符串的构造函数和拷贝构造函数可能会抛出异常
assert(!mp_eval(has_nothrow_constructor<std::string>));
assert(!mp_eval(has_nothrow_copy<std::string>));

//验证各个类型的字节对齐
assert(mp_eval(alignment_of<char>) == 1);
assert(mp_eval(alignment_of<string>) == 4);
assert(mp_eval(alignment_of<std::vector<int> >) == 4);
```

1.2.5　元数据之间的关系

type_traits 库提供四个元函数计算元数据之间的关系，它们都是两参值元函数，使用::value 返回 bool 类型的检查结果：

- is_same<T, U> ：检查 T 和 U 是否是相同的类型；
- is_convertible<From, To> ：检查 From 是否可隐式转型为 To 类型；
- is_base_of<Base, Derived>：检查 Base 是否是 Derived 的基类，或者两者相同。
- is_virtual_base_of<Base, Derived>：检查 Base 是否是 Derived 的虚基类。

示范这四个元函数用法的代码如下：

```
mp_data int meta_data1;                              //定义两个元数据
mp_data short meta_data2;

assert((is_same<int, meta_data1>::value));           //int 与 meta_data1 相等
assert((is_convertible<meta_data2, int>::value));    //short 可转换为 int

assert((is_base_of<string, string>::value));         //string 自身比较，是基类
assert((is_base_of<ios_base, ostream>::value));      //IO 流继承体系

//IO 流继承体系可参见 8.1.1 小节
assert((!is_virtual_base_of<ios_base, ostream>::value));
assert((is_virtual_base_of<basic_ios<char>, ostream>::value));
```

这四个元函数中最常用的是 is_same<>，它被用于模板元编程中比较两个元数据是否相等。

1.2.6 元数据转换

之前我们看到的元函数都是值元函数，它们返回 bool 值或者整数，下面元函数将会真正开始对元数据（类型）进行计算，输入一个类型然后输出一个新的类型，随意地处理 C++ 的类型，从中可以初步体会元编程中**类型计算**的威力和魅力。

元函数处理类型的实际转换规则比较复杂，这里仅给出较一般的情形，更精确的转换规则请读者参考 Boost 文档。

基本的元数据“加法”

这些“加法”元函数为元数据 T 添加 const、volatile、指针和引用等修饰，返回变

动后的新类型：

- add_const<T> ：返回 T const；
- add_volatile<T> ：返回 T volatile；
- add_cv<T> ：返回 T const volatile；
- add_pointer<T> ：返回 T*；
- add_reference<T> ：返回 T&；
- add_lvalue_reference<T>：对于对象类型返回左值引用，通常是 T&，否则返回 T；
- add_rvalue_reference<T>：对于对象类型返回右值引用，通常是 T&&，否则返回 T。

示范这些元函数用法的代码如下：

```
mp_data mp_exec(add_const<int>) mdata1;                //添加 const 修饰
assert((is_const<mdata1>::value));                     //新元数据是常类型
assert((is_same<mdata1, int const>::value));           //比较相等

mp_data mp_exec(add_pointer<double>) mdata2;           //添加指针修饰
assert((is_pointer<mdata2>::value));                   //新元数据是指针类型
assert((is_same<mdata2, double*>::value));             //比较相等

mp_data mp_exec(add_reference<mdata2>) mdata3;         //添加引用修饰
assert((is_reference<mdata3>::value));                 //新元数据是引用类型
assert((is_same<mdata3, double*&>::value));            //是指针的引用类型
```

基本的元数据“减法”

与元数据的“加法”操作相反，“减法”操作可以移除元数据 T 的 const、volatile、指针和引用等修饰，返回变动后的新类型：

- remove_const<T> ：移除 T 的顶层 const 修饰；
- remove_volatile<T> ：移除 T 的顶层 volatile 修饰；
- remove_cv<T> ：移除 T 的顶层 const 和 volatile 修饰；

- remove_pointer<T> ：移除 T 的指针修饰（*）；
- remove_reference<T>：移除 T 的引用修饰（&）。

示范这些元函数用法的代码如下：

```
mp_data int const **& mdata1;                          //一个指针的指针的引用类型

mp_data mp_exec(remove_pointer<mdata1>) mdata2;        //移除指针修饰
assert((is_same<mdata2, mdata1>::value));              //因为类型是引用所以不改变

mp_data mp_exec(remove_reference<mdata2>) mdata3;      //移除引用修饰
assert((is_pointer<mdata3>::value));                   //新元数据是指针类型
assert((is_same<mdata3, int const**>::value));         //比较相等

mp_data mp_exec(remove_pointer<                //连续移除两个指针，元函数嵌套调用
      mp_exec(remove_pointer<mdata3>)>) mdata4;
assert((is_const<mdata4>::value));                     //新元数据是常类型
assert((is_integral<mdata4>::value));                  //新元数据是整型
assert((is_same<mdata4, int const>::value));           //比较相等

mp_data mp_exec(remove_const<mdata4>) mdata5;          //移除 const 修饰
assert((is_same<mdata5, int>::value));                 //新元数据是整型
```

这段代码中我们首先要注意第一个元函数的使用，因为元数据 mdata1 是一个引用类型而不是指针，所以元函数 remove_pointer<>无效，直接返回元数据本身。接下来在移除顶层（最右边）的引用修饰后又连续调用了两次元函数 remove_pointer<>，这样就成功消去了 int const 的两个指针修饰，得到了不含引用和指针修饰的实际类型。

处理算术类型

下列的五个元函数用于处理算术类型元数据（is_arithmetic<T>::value == true），但不能处理 bool 类型：

- make_signed<T> ：返回 T 相应的有符号整数类型，cv 修饰不变；
- make_unsigned<T> ：返回 T 相应的无符号整数类型，cv 修饰不变；
- integral_promotion<T> ：返回 T 的右值整型提升结果，如 short 提升为 int，cv 修饰不变；

- floating_point_promotion<T>：返回 T 的右值整型提升结果，如 float 提升为 double，cv 修饰不变；
- promote<T>：返回 T 的右值算术类型提升结果，相当于 integral_promotion<T> 或 floating_point_promotion<T>。

示范这些元函数用法的代码如下：

```
mp_data short const mdata1;                          //定义元数据

mp_data mp_exec(make_signed<mdata1>) mdata2;         //添加符号
assert((is_const<mdata2>::value));                   //常量性不变
assert((is_signed<mdata2>::value));                  //有符号

//因为 mdata1 原本就是有符号数，所以两者相等
assert((is_same<mdata2, signed short const>::value));

mp_data mp_exec(make_unsigned<mdata2>) mdata3;       //去除符号
assert((is_same<mdata3, unsigned short const>::value));

mp_data mp_exec(integral_promotion<mdata3>) mdata4;  //提升整数类型
assert((is_same<mdata4, int const>::value));         //变为 int，符号修饰消失

mp_data mp_exec(floating_point_promotion<float>) mdata5;     //提升类型
mp_data mp_exec(promote<mdata5>) mdata6;             //再次提升类型
assert((is_same<mdata5, mdata6>::value));            //浮点最大类型就是 double
```

处理数组类型

type_traits 库处理数组类型主要是操作它的维度，共有四个元函数：

- remove_extent<T>　　：移除数组的最顶层维度（降低一个维度）；
- remove_bounds<T>　　：同 remove_extent<T>；
- remove_all_extents<T>：移除数组的所有维度（变为 0 维的普通类型）；
- decay<T>　　：先执行 remove_reference<T>得到元数据 U，如 U 为数组类型，则返回 remove_extent<U>*，若 U 为函数类型，则结果为 U*，否则返回 U。通常来说，decay<T>最后得到的是一个降维后的类型指针[①]。

① decay 的英文含义是“朽化”、“腐化”。

这些元函数只能操作数组，对于非数组类型则不发生改变，示范代码如下：

```
mp_data int(mdata1)[5][7][9] ;                      //多维数组元数据的定义比较特殊

mp_data mp_exec(remove_extent<mdata1>) mdata2;        //移除顶层维度
assert((is_same<mdata2, int[7][9]>::value));          //比较相等

mp_data mp_exec(remove_bounds<mdata1>) mdata3;        //移除顶层维度
assert((is_same<mdata3, mdata2>::value));             //与remove_extent<>同效果

mp_data mp_exec(remove_all_extents<mdata1>) mdata4;   //移除所有维度
assert((is_same<mdata4, int>::value));                //得到数组元素类型

mp_data mp_exec(decay<mdata2>) mdata5;                //获得降维的数组指针
assert((is_same<mdata5, int(*)[9]>::value));          //注意数组指针类型的写法
```

1.2.7 解析函数元数据

解析函数元数据的元函数 function_traits<>属于非标准元函数，它能够返回多个值，包括函数的参数数量，参数类型和返回类型，支持解析最多 10 个参数的函数。

function_traits<T>要求输入的元数据（类型）必须满足 is_function<T>::value == true，不能是函数指针或者引用。如果输入的不是函数类型，那么可以用元函数 remove_reference<T>、remove_pointer<T>来转换类型，否则会导致编译错误（即元程序执行失败）。

function_traits<>的类摘要如下：

```
template <class T>
struct function_traits
{
   static const std::size_t  arity;          //返回函数的参数数量
   typedef some-define       result_type;    //返回函数的返回值类型
   typedef some-define       argN_type;      //返回函数第N个参数的类型
};
```

因为 function_traits<>不是标准元函数，所以不能使用宏“伪关键字”来调用，必须使用域操作符直接写出内部类型定义，示例代码如下：

```
mp_data void(mdata1)(int, string);                    //注意函数类型的定义方式
assert((is_function<mdata1>::value));                 //验证函数类型元数据
```

```
const size_t n = function_traits<mdata1>::arity;     //获得函数参数数量
assert((n == 2));

mp_data function_traits<mdata1>::result_type rtype;  //函数的返回类型
assert((is_void<rtype>::value));

mp_data function_traits<mdata1>::arg2_type a2type;   //第二个参数的类型
assert((is_same<a2type, string>::value));
```

某种程度上说，function_traits<>::result_type 的效果与 boost::result_of<>::type 相同，但 function_traits<>只能处理函数类型，而 boost::result_of<>可以处理任意的可调用类型——函数、函数指针和函数对象。

10.4 小节的 function_types 库实现了功能更强大的函数元数据处理功能，读者可进一步参考。

1.2.8 实现原理

type_traits 库里的元函数虽然功能都比较简单，但其实现却比较复杂，而且还使用了预处理元编程和一些特别的技巧，这里仅以较简单的值元函数 is_integral<>为例阐述其实现原理。

integral_constant<>

type_traits 库的许多值元函数都使用了元函数转发技术，把元参数转发给元函数 integral_constant<>进行计算[①]，也就是说 integral_constant<>是大多数值元函数的 public 基类。

integral_constant<>的类摘要如下（有简化）：

```
template <class T, T val>                           //计算类型为 T，值为 val 的整数
struct integral_constant
{
   typedef integral_constant<T, val>  type;      //定义自身为返回元数据
   typedef T  value_type;                           //返回值的类型
   static const T value = val;                      //以::value 返回整数值 val
};
```

① 实际上 integral_constant<>又将元参数转发给了元函数 mpl::integral_c<>，见 11.2 小节。

顾名思义，integral_constant<>是计算整型常数的元函数，它以::type 返回自身作为元函数的计算结果，::value 返回整型常数，相当于把整数 val 做了一层元函数包装。

integral_constant<>可以这样使用：

```
//元函数计算 int 型整数 10
assert((integral_constant<int, 10>::value == 10));
//元函数计算 char 型整数 0x30，使用 Boost 静态断言
BOOST_STATIC_ASSERT((integral_constant<char, 0x30>::value == '0'));
//元函数计算 short 型整数 100
BOOST_STATIC_ASSERT((integral_constant<short, 100>::value == 100));
```

因为 type_traits 库中的大部分值元函数的计算结果是 bool 值，因此 type_traits 库特别又提供了两个针对 bool 元数据特化的无参元函数 true_type 和 false_type，它们的声明如下：

```
typedef integral_constant<bool, true > true_type;
typedef integral_constant<bool, false> false_type;
```

这两个元函数总返回 true 或者 false：

```
//直接使用::value 获得计算结果
BOOST_STATIC_ASSERT(true_type::value == true);
//使用模板元编程“伪关键字”mp_eval 获得计算结果
BOOST_STATIC_ASSERT(mp_eval(false_type) == false);
```

is_integral<>

is_integral<>的实现使用了模板特化技术，对于非整数的类型元函数总是以::value 返回 false，实现代码如下[1]：

```
template< typename T >
struct is_integral :
    boost::integral_constant<bool, false> //元函数转发，返回 false
{};
```

随后 is_integral<>对 bool、char、unsigned int、signed int 等十余个整数类型进行了模板特化，其手法与 1.1.5 小节的例子完全相同，例如：

① 实际的 is_integral<>代码使用了预处理元编程和复杂的条件编译，这里的代码做了适当的简化。

```
template<>
struct is_integral<bool> :                      //对 bool 类型特化
    boost::integral_constant<bool, true>        //元函数转发，返回 true
{};
template<>
struct is_integral<char> :                      //对 char 类型特化
    boost::integral_constant<bool, true>        //元函数转发，返回 true
{};
...//更多的整数类型特化
```

通过这样的模板特化，编译器在计算（实例化）is_integral<>元函数的时候就可以实现根据参数分别处理：对于整数类型执行特化形式返回 true，其他的类型则返回 false。

type_traits 库里的其他值元函数实现基本与 is_integral<>类似，如果想要更深入研究它们的工作原理就需要学习 mpl 元编程库（第 11 章）了。

1.2.9 应用示例

本小节将用 type_traits 库提供的元函数改写 1.1.5 小节的 demo_func<>元函数，帮助读者进一步熟悉 type_traits 库和模板元编程。

这次我们不使用模板特化，而是使用元函数转发调用 mpl 库的分支元函数 if_<>，它类似 if-else 语句，可以根据第一个元数据的真假来选择后两个元数据之一（参见 11.3 节）：

```
mp_arglist<mp_arg T>
mp_function demo_func:                          //使用元函数转发
  mpl::if_<is_pointer<T>,                       //根据是否是指针类型执行分支
      typename add_const<                       //第二步：添加 const 修饰
          typename remove_pointer<T>::type      //第一步：移除指针操作符
          >::type,                              //返回 const T
      typename add_pointer<                     //第二步：添加指针操作符
          typename add_const<T>::type           //第一步：添加 const 修饰
          >::type                               //返回 const T*
  >{};                                          //元函数结束
```

使用 type_traits 元函数的实现明显比直接的模板特化实现要复杂些，它充分展现了模板元编程的函数式本质，程序的实现都是通过函数的嵌套调用完成的，程序员需要在自己的头脑中维护一个“函数的堆栈”才能弄清楚它们的调用过程。

使用另一个 mpl 元函数 eval_if<>也可以达到同样的效果，它会自动计算后两个元函数的结果，不必再写::type，代码可以得到一点简化：

```
mp_arglist<mp_arg T>
mp_function demo_func:                        //使用元函数转发
  mpl::eval_if<is_pointer<T>,                 //根据是否是指针类型执行分支
      add_const<                              //直接写元函数，不需要用 typename
          typename remove_pointer<T>::type    //这里还需要返回元数据
          >,                                  //不需要写::type
      add_pointer<                            //直接写元函数，不需要用 typename
          typename add_const<T>::type         //这里还需要返回元数据
          >                                   //不需要写::type
  >{};//元函数结束
```

1.3 总结

把较深奥的模板元编程作为全书的第一章确实有点冒险，但对于本书来讲的确有必要，感谢读者能够读到这里。

本章首先介绍了模板元编程的基础知识，介绍了元编程/元程序、元数据、元函数和元函数转发等基本概念。元编程是一种超越普通程序的程序，在 C++中元编程是使用模板技术实现的，所以它又被称为模板元编程，可以由 C++编译器在编译期解释执行，把部分计算量由运行时转移到编译时完成，提高程序的运行效率。

元数据是元编程的操作对象，可以是整数（含 bool）或任意的 C++类型；元函数是元编程的核心，它表现为一个 C++模板类，我们必须使用元函数才能操作元数据，它以内部定义::type 或::value 返回计算的结果，并且可以使用 public 继承的方式实现元函数转发。这种函数式的计算方式初接触会觉得有些古怪，但只要我们理解了它操作类型的本质就能够逐渐掌握用法，如果用起来不顺手还可以使用宏来定义“伪关键字”减轻不适应感。

理解了这些模板元编程基本概念后我们还可以深入考察其他既有的库，它们实际上已经或多或少地实践了模板元编程的理念，例如：

- std::tr1::result_of：它能够推导出可调用类型的返回值，以 result_of<T>::type 的形式给出，因此它就是一个元函数。
- boost::ref 的 unwrap_reference<T>：它能够解开 boost::reference_wrapper 的包装。

- boost::call_traits（10.2 小节）：它能够推导出最合适的调用参数类型。
- …………

这样的例子还有很多，它们的类型“推导”实际上就是模板元编程中元函数对元数据（类型）的计算[①]。

traits 是模板元编程中的一个非常重要的概念，它可以萃取类型中的许多重要信息，利于我们在编译期提早做出决断。本章我们重点讨论了 type_traits 库，它可以萃取类型的基本（但不是所有）信息，还顺便完整地探讨了 C++的类型系统。之后我们还会看到其他的 traits 库，如萃取迭代器信息的 iterator_traits（3.3 小节）和萃取函数信息的 function_types（10.4 小节）。

type_traits 库提供了大量有用的元函数，可以检查元数据的类型和关系，提取它的各种属性，也可以增加 const、volatile 等类型修饰词，使用它能够丰富我们的编程词汇，更精确地描述 C++的类型系统，配合静态断言可以极大地保证程序的正确性。不过 type_traits 库中有的元函数需要编译器的支持才能正常工作，也就是说不是所有的 C++编译器都支持 type_traits 的所有功能，但对于大多数现代编译器应该都不成问题。

接下来的章节中我们不会立即开始模板元编程，但会使用模板元编程的概念来阐述 Boost 程序库，会看到相当多的 type_traits 库和元编程的应用展示。当然，如果读者感兴趣，也可以直接阅读第 11 章，连续完整地学习模板元编程。

① 等读者阅读完第 11 章了解了 mpl 库后再看这些 Boost 组件相信就会有种“豁然开朗”的感觉了。

第2章

实用工具

本章将使用第 1 章的元编程知识研究一些功能比较简单，但实现原理却涉及 C++语言深层次概念细节的 Boost 组件。

首先来讨论看似无用的“空类”，然后研究两个“智能操作符”check_delete 和 addressof，再讨论初始化问题和类型转换，最后是指针和 RAII。这些小工具使用了特别的技巧，都有着近乎于魔术般的神奇功能，希望读者阅读完本章后能够对 C++有更深入的认识。

2.1 compressed_pair

compressed_pair 库提供一个与 std::pair<> 非常相似的模板类 compressed_pair<>，同样能够容纳任意两个元素，但它针对空类成员进行了特别的优化，可以“压缩”pair 的大小。

compressed_pair 位于名字空间 boost，为了使用 compressed_pair 组件，需要包含头文件<boost/compressed_pair.hpp>，即：

```
#include <boost/compressed_pair.hpp>
using namespace boost;
```

2.1.1 什么是空类

compressed_pair 与 std::pair 非常相似，不同之处是它对“空类”成员做了特殊的处理，所以，我们首先要了解什么是空类。

所谓的“空类”是一个 class/struct 类型，它没有非静态成员变量（静态成员不会

增加类实例大小），也没有虚拟函数（会导致虚表指针），使用 type_traits 库的元函数表述就是 is_class<T> && is_empty<T>。

下面的代码定义了一个最简单的空类：

```
class empty1{};   //最简单的空类
```

乍看起来空类似乎没有什么用。的确，像 empty1 这样仅有类声明而没有任何其他东西的真正的“空类”确实没有用[①]，但还有另外许多很有用的“空类”：它们被编译器认为是“空类”，但实际上却不是“空类”，可以含有许多有用的功能，但类实例（instance）却不会占用内存空间——这些功能包括 typedef、静态成员变量、成员函数、友元函数、枚举等，起到一个对功能“打包”的作用。

例如下面的几个“空类”，它们虽然不含普通数据成员变量，但能够用在很多地方：

```
class empty2                //空类，含有一个静态成员变量
{
  static const int MAX = 100;
};
class empty3                //空类，含有成员函数
{
public:
  void print()
  {   cout << "this is a empty class." << endl;}
                            //还可以有其他的非虚成员函数
};
```

C++现实代码中有非常多的有用“空类”的例子，如标准库中的函数对象 greater<>、less<>、plus<>等，它们都是“空类”，因为它们仅提供一个 operator()。许多 Boost 组件也都使用了大量作为技术手段的“空类”，例如 integer 库和 operators 库，还有模板元编程中使用的所有元函数。

虽然空类不含有任何数据成员，理论上不应占据内存，但实际上单独使用它时仍然需要占用一定的内存空间。因为 C++不允许存在 0 大小的对象，所以大多数 C++编译器会暗地里在类中插入一个 char 以使它具有至少为 1 字节的大小[②]，这样一来下面的断言通常总成立：

① 纯粹的空类也不一定完全无用，在模板元编程中它可以作为类别的 tag 标记（如迭代器分类，见 3.1.2 小节），或者是特殊的“哨兵”角色（如 tuple）。

② 空类的大小依据编译器的行为而不同，有的编译器可能会令空类增大至 sizeof(int)。

```
assert(sizeof(empty1) == 1);              //空类的实例占用 1 字节内存
assert(sizeof(greater<int>) == 1);        //空类的实例占用 1 字节内存
```

因此，如果我们编写的类含有类型为空类的成员变量，通常会导致增加一点点原本不应该存在的空间开销，对于 std::pair 这种简单的结构来说特别明显。

compressed_pair 正是为解决这个问题而来，它使用了模板元编程技术（type_traits、call_traits）和多重继承来优化空类成员，可以“压缩” pair 的大小以节约空间。

2.1.2 类摘要

compressed_pair 是一个模板类，摘要如下：

```
template <class T1, class T2>
class compressed_pair
{
public:
   typedef T1   first_type;                     //第一个成员类型定义
   typedef T2   second_type;                    //第二个成员类型定义
   ...                                          //其他 typedef

   compressed_pair();                           //四种构造函数
   compressed_pair(first_param_type x, second_param_type y) ;
   explicit compressed_pair(first_param_type x);
   explicit compressed_pair(second_param_type y) ;

   first_reference         first();             //返回第一个成员
   first_const_reference   first() const ;
   second_reference        second() ;           //返回第二个成员
   second_const_reference second() const ;

   void swap(compressed_pair& y) ;
};
```

compressed_pair 同 std::pair 一样，接受任意两个类型数据作为它的模板参数①，

① 请读者注意：使用 union 类型作为 compressed_pair 的模板参数需要编译器的支持，因为 compressed_pair 依赖于 type_traits 库中元函数 is_union<>的能力。就目前作者所知，VC8、VC9 和 GCC4.x 都可以，但 VC6、GCC3.x 不可以，并且 VC6 还存在一个对空类成员赋值的 bug。

并使用 typedef 定义了若干内部类型定义方便使用（即 traits），对于两个模板参数是同一类型（boost::is_same<T1, T2>::value == true）的情况还进行了模板偏特化。

compressed_pair 不提供任何比较操作符重载，因此不能像 std::pair 一样在两个 compressed_pair 对象间执行比较操作，但这不是什么太大的问题，可以很容易地自行解决（参见 2.1.6 小节）。

2.1.3 构造与赋值

compressed_pair 提供了以下四种形式的构造函数：

- 无参数的构造函数将缺省构造两个成员。如果成员是一个 POD 类型，那么不会被自动赋初值。
- 单参数的构造函数将自动推导类型，把参数赋值给恰当的成员。如果两个成员的类型相同，那么参数会同时赋值给两个成员。
- 两参数的构造函数将使用参数分别初始化两个成员，行为与 std::pair 类似。

compressed_pair 的这四种形式的构造函数可以适应各种情况下的对象创建，较 std::pair 要么不指定初值、要么全指定初值的方式更为灵活。下面的代码示范了这四种构造函数的用法：

```
//无参构造，第一个成员值不确定，第二个成员缺省构造为空串
compressed_pair<int, string> cp1;
//单参构造，第一个成员有初值 10，第二个成员缺省构造为空串
compressed_pair<int, string> cp2(10);
//单参构造，第一个成员值不确定，第二个成员有初值"hello"
compressed_pair<int, string> cp3("hello");
//双参构造，第一个成员有初值 20，第二个成员有初值"pair"
compressed_pair<int, string> cp4(20, "pair");
```

compressed_pair 也支持拷贝构造和赋值操作：

```
compressed_pair<int, string> cp5(cp2);    //拷贝构造
cp1 = cp4;                                //赋值操作
```

因为使用了 call_traits 库（参见 10.2 小节）传递构造函数的参数，所以 compressed_pair 的模板参数可以是引用，这是对 std::pair 的增强——std::pair 不允许成员为引用类型，会导致“引用的引用”的错误，例如：

```
int i = 313;
string s;

compressed_pair<int&, string&> cp(i, s); //正确
std::pair<int&, string&> p(i, s);        //编译错误
```

2.1.4 用法

compressed_pair 访问成员需要使用函数形式的 first()和 second()，它们返回 compressed_pair 内部成员的引用，这与 std::pair 直接访问内部成员的方式不同。但除了要加上 operator()之外，compressed_pair 的访问成员用法与 std::pair 的 first/second 几乎相同：

```
compressed_pair<int, string> cp;      //缺省构造
assert(cp.second().empty());          //第二个成员是空串

cp.first() = 10;                      //可以当做左值被赋值
cp.second() = "hello";
assert(10 == cp.first());             //也可以当做右值
assert(!cp.second().empty());         //调用成员的成员函数
```

compressed_pair 容纳非空类时与 std::pair 的差别不大，不过当 pair 的两个成员中存在空类时就能够起到“压缩存储空间”的优化效果，见下面的代码示例：

```
assert(sizeof(compressed_pair<int, empty1>) == sizeof(int));
assert(sizeof(std::pair<int, empty1>) > sizeof(int));
cout << sizeof(std::pair<int, empty1>)  << endl;

cout << sizeof(compressed_pair<empty1, empty2>) << endl;
cout << sizeof(std::pair<empty1, empty2>) << endl;
```

这段代码的前三行定义了具有一个空类的 compressed_pair 和 std::pair，compressed_pair 只有 sizeof(int)的大小，空类被优化成了不占用空间的真正“空类”，而 std::pair 则因为字节对齐的原因大小变为两个 sizeof(int)的大小。

代码的后两行定义了成员全为空类的 compressed_pair 和 std::pair，这时 compressed_pair 的大小为 1，有一个空类的存储被优化掉了，而 std::pair 的大小为 2。

因此，如果用户对内存空间的使用十分敏感，而且程序中可能出现大量的空类的时候，就可以使用 compressed_pair。

2.1.5 实现原理

compressed_pair 的名字可能会给人以误解，误以为它使用了什么玄妙的压缩算法，而实际上，它并没有做任何的“压缩”动作，而是利用了编译器的空基类优化技术（empty base-class optimisation，简称 EBO），也许名字叫 optimized_pair 更加恰当。

compressed_pair 库在 boost::details 子名字空间下定义了一个模板类 compressed_pair_imp，它是 compressed_pair 的真正实现，被用于私有继承，其声明如下：

```
template <class T1, class T2, int Version>
class compressed_pair_imp;
```

库根据模板参数类型是否相同和两个成员是否为空这两个条件共定义了 2×3=6 个偏特化的 compressed_pair_imp 实现，再使用 type_traits 库的 is_same<>、is_empty<>、remove_cv<>以及 details 子名字空间里的模板元函数 compressed_pair_switch<>进行编译期元计算，决定 int 模板参数 Version 的值，从而最终决定使用哪个版本的 compressed_pair_imp。

由于偏特化的 compressed_pair_imp 已经知道了模板参数是否为空类，因此它就从空类 protected 继承，而不是作为成员变量，这样编译器就可以使用空基类优化技术来优化类的大小。

例如，对于第一个成员（T1）是空类的情况，compressed_pair 将使用偏特化的 compressed_pair_imp<T1,T2,1>，其实现摘要如下：

```
template <class T1, class T2>
class compressed_pair_imp<T1, T2, 1>              //特化版本 1，T1 是空类
   : protected ::boost::remove_cv<T1>::type       //从 T1 继承
{
public:
   compressed_pair_imp(first_param_type x, second_param_type y)
      : first_type(x), second_(y) {}

   first_reference      first()      {return *this;}  //注意这里
   second_reference      second()      {return second_;}
private:
   second_type second_;                           //只有第二个类型的成员变量
};
```

在这种情况下，compressed_pair 将只有一个成员变量，空基类 T1 原本所需的至少为 1 的存储空间将被编译器优化掉。

因为使用了继承来实现 compressed_pair，"空类"不是以成员变量的形式存在，所以 compressed_pair 不能像 std::pair 那样直接访问成员，只能通过函数以引用的方式来间接访问所谓的"成员"。

2.1.6 功能扩展

compressed_pair 与 std::pair 在名字和用法上都十分相似，但 compressed_pair 还缺少类似 std::make_pair() 的辅助函数和比较操作，如果需要，我们也可以自己实现。

make_compressed_pair

仿照 std::make_pair() 可以很容易地实现工厂函数 make_compressed_pair()，代码如下：

```
template<typename T1, typename T2> inline
compressed_pair<T1, T2>
make_compressed_pair(T1 t1, T2 t2)
{
  return compressed_pair<T1, T2>(t1, t2);
}
```

make_compressed_pair() 很容易使用，配合 boost.typeof 库的 BOOST_AUTO 宏将使 compressed_pair 的创建更简单，无须指定模板参数，例如：

```
BOOST_AUTO(cp1, make_compressed_pair(10, "char*"));
BOOST_AUTO(cp2, make_compressed_pair(3.14, empty1()));
```

实现比较功能

使用 operators 库（详见推荐书目[1]），可以使用继承的方式实现 compressed_pair 的比较功能①，这要求 pair 的两个成员都是可比较的——确切地说，应该要求非空类是可比较的，空类应该总是相等的。作为示例，本书仅使用 equality_comparable 概念实现了相等操作：

① 这里的 compressed_pair_ex 没有处理 T1==T2 的特化，因此不能容纳两个相同的类型，请读者注意。

```
template<typename T1, typename T2>
class compressed_pair_ex:
  public boost::compressed_pair<T1, T2>,             //继承 compressed_pair
  boost::equality_comparable<compressed_pair_ex<T1, T2> >    //相等概念
{
public:
  typedef compressed_pair_ex<T1, T2> this_type;
  typedef compressed_pair<T1, T2>base_type;

  //四种形式的构造函数
  compressed_pair_ex() : base_type() {}
  compressed_pair_ex(first_param_type x, second_param_type y) :
          base_type(x, y) {}
  explicit compressed_pair_ex(first_param_type x) : base_type(x) {}
  explicit compressed_pair_ex(second_param_type y) : base_type(y) {}

  //简单地执行比较操作，存在缺陷，无法比较空类
  friend bool operator==(const this_type& l, const this_type& r)
  {
      return l.first() == r.first() &&
              l.second() == r.second() ;
  }
};
```

上面的代码定义了一个 compressed_pair 的子类 compressed_pair_ex，它同时又是 boost::equality_comparable 的子类，因此可以执行相等或不等的操作，但 operator==的实现并没有处理空类的情况，真正对 compressed_pair 实现完整的比较操作需要使用 type_traits 库里的元函数，仿造 compressed_pair 的实现用元编程做模板特化处理。

元编程实现比较功能

在这里，我们将模仿 compressed_pair 的实现方式，使用 compressed_pair_switch<>计算版本，然后用一个私有模板成员函数特化来进行分支判断：

```
private:                    //以下均为 compressed_pair_ex 的私有成员函数
  template<int Version>     //仅为特化而使用，无需返回值
  bool _compare(const this_type& r) const{}

  template<>                //都不是空类
```

```
bool _compare<0>(const this_type& r) const
{
    return this->first() == r.first() &&
        this->second() == r.second() ;
}
template<>      //T1 是空类
bool _compare<1>(const this_type& r) const
{
    return this->second() == r.second() ;
}
template<>      //T2 是空类
bool _compare<2>(const this_type& r) const
{
    return this->first() == r.first() ;
}
template<>      //T1、T2 都是空类
bool _compare<3>(const this_type& r) const
{
    return true ;
}
//T1、T2 相同的 4、5 情况在这里不需要处理
```

比较操作符调用 compressed_pair_switch<>计算版本，根据版本再决定调用哪个_compare()特化形式，重新实现如下：

```
friend bool operator==(const this_type& l, const this_type& r)
{
  return l._compare<
      boost::details::compressed_pair_switch
      <T1, T2,
       boost::is_same<typename boost::remove_cv<T1>::type,
          typename boost::remove_cv<T2>::type
          >::value,                 //T1、T2 是否相同
       boost::is_empty<T1>::value,    //T1 是否为空类
       boost::is_empty<T2>::value     //T2 是为空类
       >::value                     //得到元函数计算结果
      >(r);                         //完成元计算，调用特化的成员函数
}
```

现在 compressed_pair_ex 就能够完全地支持空类和非空类的比较操作了，可以这样使用：

```
compressed_pair_ex<int, double> cp1(10,0), cp2(10,0);
assert (cp1 == cp2);

compressed_pair_ex<int, empty1> cp3(0), cp4(10);
assert(cp3 != cp4);

compressed_pair_ex<empty1, empty2> cp5, cp6;
assert (cp5 == cp6);
```

另一种比较功能的实现方法

如果不想使用继承方式扩展功能，我们也可以直接在 boost 名字空间为 compressed_pair 定义比较操作：

```
namespace boost   //在 boost 名字空间里增加比较操作，元编程版本可仿造实现
{
  template<typename T1, typename T2>
  bool operator==(const compressed_pair<T1, T2>& l,
      const compressed_pair<T1, T2>& r)
  {
      return l.first() == r.first() &&
          l.second() == r.second() ;
  }
}
```

这样一来我们就无须使用专门的类 compressed_pair_ex，可以直接比较 compressed_pair，只是这种做法就不能利用 operators 库的好处，必须要写出所有所需操作符的实现，代码上要冗余一些。

2.2 checked_delete

checked_delete 是对 C++关键字 delete 的增强，可以在编译期保证 delete 或 delete[]删除的是一个指向“完整类”的指针，避免在运行时发生未定义行为，是一个更加“智能”的 delete。

checked_delete 位于名字空间 boost，为了使用 checked_delete 组件，需要包含头文件<boost/checked_delete.hpp>[①]，即：

```
#include <boost/checked_delete.hpp>
```

① 也可以直接包含<boost/utility.hpp>，它内含数个小工具的实现。

```
using namespace boost;
```

2.2.1 函数的用法

checked_delete 库包含两个函数和两个函数对象，分别是：

- checked_delete()/checked_deleter ：用于删除普通指针；
- checked_array_delete()/checked_array_deleter：用于删除数组指针，相当于 delete[]。

首先我们来了解函数形式的用法，模板函数 checked_delete()和 checked_array_delete()的声明如下：

```
template<class T> void checked_delete(T * p);
template<class T> void checked_array_delete(T * p);
```

正如名字所表达的，它们是“进行检查的 delete 操作”，基本功能等价于 delete 和 delete[]，用法也完全相同，只是我们需要把操作指针变量的 delete 表达式改成函数调用式。例如：

```
int *p1 = new int(10);                    //普通指针
checked_delete(p1);                       //删除普通指针

int *p2 = new int[10];                    //数组指针
checked_array_delete(p2);                 //删除数组指针
```

注意，checked_delete()必须用于删除普通指针，而 checked_array_delete()必须用于删除数组指针，千万不可误用，它们还没有智能到能够识别普通指针和数组指针的程度（这也是为什么使用两个函数的原因），正确地使用它们还是程序员的责任。

除了 int、double 等基本类型，checked_delete()和 checked_array_delete()当然也可以用来删除对象指针：

```
class demo_class                          //一个简单的空类
{
public:
  ~demo_class()                           //析构函数，输出提示信息
  {   cout << "dtor exec." << endl;   }
};
```

```
int main()
{
  demo_class *p1 = new demo_class;          //对象指针
  checked_delete(p1);                       //删除对象指针

  demo_class *p2 = new demo_class[10];      //对象指针数组
  checked_array_delete(p2);                 //删除对象数组指针
}
```

这段代码中我们定义了一个简单的类，它带有“非平凡”的析构函数（non-trivial destructor），使用 checked_delete 会正确地删除它们，程序运行后会在控制台输出 11 行“dtor exec.”。

2.2.2 函数对象的用法

模板类 checked_deleter 和 checked_array_deleter 的声明如下：

```
template<class T>
struct checked_deleter
{
    typedef void result_type;
    typedef T * argument_type;
    void operator()(T * x) const;
};
template<class T>
struct checked_array_deleter
{
    typedef void result_type;
    typedef T * argument_type;
    void operator()(T * x) const;
};
```

checked_deleter 和 checked_array_deleter 重载了 operator()，因而可以像函数一样被调用，它们实际上仅是对同名函数加上了简单的类包装。

因为它们是函数对象，不具备自动推导模板参数的能力，因此在使用时必须用模板参数指明要删除的对象类型，否则会无法通过编译：

```
demo_class *p1 = new demo_class;
checked_deleter<demo_class>()(p1);          //删除对象指针
```

```
demo_class *p2 = new demo_class[10];
checked_array_deleter<demo_class>()(p2);      //删除对象数组指针
```

这段代码的功能与刚才的 checked_delete() 函数调用完全相同，注意函数对象的使用方式：必须先在模板参数中指定要删除的对象类型，然后用一对圆括号调用构造函数生成一个临时函数对象，最后才能使用 operator() 调用删除功能。

表面上看函数对象的用法似乎很麻烦（实际上也确实如此），但因为它们的定义完全符合 C++98 标准规范，故可以传递给那些需要函数对象的泛型代码，例如搭配标准库算法操作容器里的指针：

```
vector<demo_class*> v;                    //一个容纳指针元素的标准容器
v.push_back(new demo_class);              //添加两个元素
v.push_back(new demo_class);

//调用 for_each 算法删除容器内的指针
for_each(v.begin(),v.end(), checked_deleter<demo_class>());
```

上面这段代码使用标准库容器 vector 保存了若干对象指针，随后使用标准库算法 for_each，传入 checked_deleter 函数对象逐个安全地删除，这比使用循环遍历容器再用 delete 删除指针要方便的多[①]。

对函数对象的改进

checked_deleter 是一个模板类，使用时必须要指定模板参数，显得有些麻烦。我们可以自定义一个类似的函数对象，它是非模板类，但有模板成员函数，所以能够自动推导模板参数：

```
struct my_checked_deleter
{
  typedef void result_type;       //返回类型定义

  template<class T>
  void operator()(T* x) const     //模板成员函数
  {
      boost::checked_delete(x);   //调用 checked_delete 函数
  }
};
```

① 代码中的 for_each 算法中我们也可以直接传递函数，但同样需要指明它的模板类型，用法如下所示：for_each(v.begin(),v.end(), checked_delete<demo_class>)。

my_checked_deleter 可以如 checked_deleter 一样工作，但省去了写模板参数的麻烦：

```
int *p = new int(10);
my_checked_deleter()(p);            //无须使用模板参数指明类型

vector<int*> v;
v.push_back(new int(10));
for_each(v.begin(), v.end(),
      my_checked_deleter());        //无须使用模板参数指明类型
```

2.2.3　带检查的删除

乍看上去似乎 checked_delete 与关键字 delete 没有什么不同，它仍然需要使用 delete，而且的确——之前代码中的 checked_delete 函数调用完全可以用关键字 delete 替换，运行结果不会有任何差异。

但 checked_delete 并不完全等同于 delete 关键字，它在 delete 操作之外增加了对不完整类型的检查，能够更好地保证代码的正确性。

所谓不完整类型（incomplete type）是指仅有声明而没有定义的类，它通常见于类的前向声明，例如：

```
class demo_class;//一个不完整类型，只有声明而没有具体定义
```

对一个不完整类执行 delete 操作会导致析构函数未被执行，引发未定义行为，见下面的示例代码：

```
class demo_class;                     //前向声明，不完整类
void do_delete(demo_class* p)         //一个简单的删除函数
{ delete p;}                          //调用 delete 操作符
class demo_class{...};                //这里是类的完整定义，同前
int main()
{
  demo_class *p = new demo_class();   //一个对象指针
  do_delete(p);                       //将导致未定义行为
}
```

这段代码在 VC8 下编译时会引发一个警告[①]，信息如下：

① 但不是所有的编译器都会对这段代码产生警告。

```
warning C4150: deletion of pointer to incomplete type 'demo_class'; no destructor
called
```

调用 do_delete()删除指针时不会调用 demo_class 的析构函数,因为 demo_class 是一个前向声明的不完整类，此时编译器还不知道它的析构函数。程序运行后没有任何输出，因为析构函数没有被调用，如果 demo_class 是一个具有复杂内部结构的类，那么就会可能导致资源未释放等未定义行为。

由于这段程序的代码量很少，所以我们能够很容易地查找到警告所在的位置并发现问题，但如果是在一个较大的工程中，头文件的引用和类的前向声明比较复杂，那么这样的警告就很容易被有意或者无意地忽略掉，或者被淹没在其他的错误与警告中无法发现。并且更有可能的是，有的编译器并不对此提出警告。

使用 checked_delete 可以避免这种微妙的问题——如果要删除的指针指向的不是一个完整类型，将引发编译错误：

```
class demo_class;                              //前向声明，不完整类
void do_delete(demo_class* p)
{ checked_delete(p);}                          //改用 checked_delete
//class demo_class 的完整定义在别处
int main()
{
  demo_class *p = (demo_class*)(1996);         //强制转换一个指针地址
  do_delete(p);                                //编译错误
}
```

上面的代码无法通过编译，通过编译器提供的错误信息，我们可以很容易地发现删除不完整类型的错误，从而消灭未定义行为。

2.2.4 实现原理

checked_delete 的原理相当的简单，其全部实现代码如下：

```
template<class T> inline void checked_delete(T * x)
{
    typedef char type_must_be_complete[ sizeof(T)? 1: -1 ];
    (void) sizeof(type_must_be_complete);
```

```
    delete x;                                    //调用 delete 操作符删除指针
}
```

它通过 typedef 定义了一个数组类型，其大小由要被删除的类型 T 确定。如果 T 是一个完整类型，那么 sizeof(T)表达式结果是一个正整数，数组大小为 1；如果 T 是一个不完整类型，那么 sizeof(T)表达式结果是 0，数组大小为-1。但 C++中数组的定义是不允许为负数的，所以会引发一个编译错误。

checked_array_delete 的实现与之类似，只不过最后的语句是 delete[] x，因为它被用于删除数组指针。

作为一个示范，也可以使用 1.2 小节介绍的 type_traits 库和 static_assert 库来实现自己的 checked_delete，代码如下：

```
template<class Pointer> inline                   //模板参数是指针类型
void my_checked_delete(Pointer p)                //一个自定义的 checked_delete
{
  BOOST_STATIC_ASSERT(is_pointer<Pointer>::value);   //要求是指针类型

  typedef remove_pointer<Pointer>::type T;           //元函数计算值类型
  BOOST_STATIC_ASSERT(is_object<T>::value);  //要求必须是可删除的对象
  BOOST_STATIC_ASSERT(!is_array<T>::value);  //要求不能是数组类型
  BOOST_STATIC_ASSERT(sizeof(T) >0 );        //同 checked_delete 的编译期断言

  delete p;                                  //删除指针
}
```

my_checked_delete() 函数的实现手法与 checked_delete 类似，它使用 BOOST_STATIC_ASSERT 宏在编译期断言模板类型 T 必须是一个完整类型，并增加了一些新的检查条件，效果与 checked_delete()相同。

不过读者需要注意的是检查数组的断言（!is_array<T>）实际作用不是很大，对于指向一维数组的指针来说移除指针修饰后的类型仍然是 T，但如果指针是一个指向多维数组的指针那么!is_array<T>可以正常工作：

```
int (*p)[2] = new int[2][2];                 //多维数组指针
my_checked_delete(p);                        //发生编译错误，静态断言生效
```

2.2.5 使用建议

checked_delete 被声明为内联函数，而且函数内部仅使用了编译期的 typedef，没有任何多余的可执行代码，因此它的性能与原始 delete 相比没有任何差异，并且它命名清楚，易于维护，所以可以完全替代 delete 关键字在任何地方使用。

checked_delete 的用法非常简单，但作为库用户的我们最好应当少使用，因为直接操作原始内存很容易发生各种难以察觉的错误。多数情形下我们应该改用智能指针，例如 boost::scoped_ptr 和 boost::shared_ptr，它们在内部调用了 checked_delete，可以自动地管理指针的生命周期，而且是异常安全的。

例如，下面的代码使用 shared_ptr 来管理对象指针：

```
shared_ptr<demo_class> sp(new demo_class);
...//任意操作，无论发生什么，指针总会被正确删除，无须使用 delete
```

如果因为某些原因不能使用智能指针，必须手工管理内存，那么就请使用 checked_delete，它可以保证在编译期就发现隐藏的错误。

2.3 addressof

addressof 是对 C++取地址操作（&）的增强，因为 C++允许重载 operator&，所以有时候 operator&会被程序员“欺骗”，但 addressof 总能够获取到操作对象的真实地址。

addressof 位于名字空间 boost，为了使用 addressof 组件，需要包含头文件 <boost/utility/addressof.hpp>[①]，即：

```
#include <boost/utility/addressof.hpp>
using namespace boost;
```

2.3.1 用法

addressof 是一个模板函数，声明如下：

```
template <typename T> inline
T* addressof(T& v);
```

① 也可以直接包含<boost/utility.hpp>，它内含数个小工具的实现。

addressof 与 checked_delete 类似，是一个“智能取地址操作符”，它的用法很简单，可以像通常的&操作符一样使用。例如：

```
int i;                                  //整数
string s;                               //标准库字符串
assert(&i == addressof(i));             //断言取地址相等
assert(&s == addressof(s));             //断言取地址相等
```

addressof 不仅可以操作普通变量，也可以操作数组和函数：

```
int a[10];
assert(&a == addressof(a));             //取数组地址
assert(a + 1 == addressof(a[1]));
assert(&sqrt == addressof(sqrt));       //取函数地址
```

对于重载了 operator&的类，直接使用&操作符会失效（因为这时实际上调用了重载的 operator&()函数），但 addressof 仍然可以获得变量的真实地址，请看下面的代码：

```
class dont_do_this                      //演示用，最好不要重载 operator&
{
public:
  int x,y;
  size_t operator&() const              //重载 operator&
  {       return (size_t)&y;  }         //返回成员变量 y 的地址
};
int main()
{
  dont_do_this d;
  assert(&d != (size_t)addressof(d));   //(1)
  assert(&d == (size_t)&d.y);           //(2)
  dont_do_this *p = addressof(d);       //(3)
  p->x = 1;
}
```

类 dont_do_this 重载了 operator&，因此对它的实例调用&操作符将不会返回实例的内存地址，而是改为调用重载的 operator&函数。dont_do_this 通过这种方式“隐藏”了自己的真正地址，而是返回了内部成员变量 y 的地址。

代码行(1)分别使用 operator&和 addressof 来获取变量的地址，因为 dont_do_this 重载了 operator&，因此&d 实际上获得的是 d.y 的地址，代码行(2)更清楚地证明了这一点。

而 addressof 则不受重载 operator&的影响，它总能获得真实的地址，因此代码行(3)是正确的。如果使用&d 来赋值指针变量 p，会得到一个编译错误，因为 operator&返回的是一个 size_t 类型，而不是地址[①]。当然我们也可以用强制类型转换把返回值转换为指针地址，但这样做得到的将是一个错误的内存位置，会导致未定义行为。

有的时候某些类甚至把 operator&声明为私有的，通常这种情况下无法使用&来获得变量的地址，但 addressof 同样不受影响。

```
class danger_class                //不允许调用 operator&的类
{
  void operator&() const;         //私有的 operator&
};
  danger_class d;
  //  cout << &d;                 //将导致编译错误，无法调用私有成员函数
  cout << addressof(d);           //正确，获得真实地址
```

2.3.2 实现原理

addressof 并没有应用什么特别的技巧，仅仅是使用了复杂的转型操作，核心实现代码如下：

```
reinterpret_cast<T*>(&const_cast<char&>(
  reinterpret_cast<const volatile char &>(v)));
```

代码中的类型 T 是 addressof 的模板类型参数，v 是 T 的一个引用。addressof 先使用了 reinterpret_cast<>转型操作符把 v 先强制解释成 char 类型，然后再重新解释成 T*类型，这样就得到了变量的真正地址。

因为多次使用了运行时转型，所以 addressof 的运行效率没有原始的 operator&效率高，但这点损失通常是微不足道的。

2.3.3 使用建议

为类重载 operator&不是一个非常明智的做法，很少有人会这样做，但这种可能性是存在的。

基于“害人之心不可有，防人之心不可无”的古训，我们在编写代码时应尽量做到不重载 operator&。如果怀疑某个类可能重载了 operator&，那么就使用 addressof 来获取

① 实际上，operator&不一定非要返回一个数值，可以是任何类型。

对象的真实地址。

作为一个库作者，他必须应对各种可能的情况，因此应当使用 addressof；而作为一个库用户和应用开发者，则应当编写合理的、规范的代码，消除 addressof 的应用可能。

2.4　value_initialized

在 C++中构造并初始化对象是一个非常重要的操作，但 C++98 标准中并没有对初始化问题给出非常明确的定义，有的时候变量可能会初始化为不确定的值。value_initialized 能够以一致的语法保证变量总能够被正确地初始化——拥有零值或者缺省值，这在编写泛型代码时特别有用。

value_initialized 位于 boost 名字空间，为了使用 value_initialized 组件，需要包含头文件<boost/utility/value_init.hpp>，即：

```
#include <boost/utility/value_init.hpp>
using namespace boost;
```

2.4.1　变量的初始化

C++98 标准中定义了零初始化和缺省初始化的概念，零初始化是指将变量赋予初值 0，缺省初始化是指对 POD 类型零初始化，而对类类型调用缺省构造函数进行初始化。例如：

```
int i(0);              //零初始化
assert(i == 0);
string s;              //缺省初始化
assert(s.empty());
```

但 C++98 标准还没有解决有关初始化的所有问题。很多 POD 类型变量如果不使用圆括号方式用初值进行初始化，那么就会有一个不确定的值：

```
int i;                 //i 的初值不确定，存在隐患
```

而且，目前 C++中初始化的语法也不完全统一，难以编写泛型的代码。最通用的泛型初始化语法是：

```
T v = T();
```

它将保证将变量 v 进行缺省初始化，但要求类型 T 是可缺省构造和可拷贝构造的。第二

个条件（可拷贝构造）在有的时候可能会无法满足，例如私有化了拷贝构造函数（可参见 boost.noncopyable）。

已经推出的 C++11 标准对这个问题给出了完整的解决方案，但在这之前，value_initialized 以库的形式达到了同样的目的。

2.4.2 initialized<T>

initialized 是一个模板类，类摘要如下：

```
template<class T>
class initialized{
public :
  initialized();                    //构造函数
  explicit initialized(T const & arg);

  operator T const &() const;       //类型转换
  operator T&();
  T const &data() const;            //获得值的引用
  T& data();
  void swap(initialized<T>& );

private :
  struct wrapper;                   //内部包装类
} ;
```

initialized 就像是一个类型 T 的包装类①，它可以保证被包装的对象是被正确初始化的。实际上，initialized 内部用一个私有的包装类 wrapper 来包装 T 类型的对象，并使用定位 new 表达式保证总是零初始化或者缺省初始化。

initialized 提供两种形式的构造函数，无参的构造函数执行缺省初始化，单参的构造函数则使用参数进行值初始化。

为了更方便使用，initialized 提供了到 T const &和 T&的类型转换操作，可以用在任何要求 T 的语境中，也可以显式用成员函数 data()来获得被包装对象的引用。

另外，initialized 还提供了泛型的自由函数 get()，它调用了成员函数 data()用来获取被包装对象的引用，可以用于编写统一的操作代码。

① 从包装对象这种用法来看，value_initialized 与 ref 库非常相似，读者可对比参考。

2.4.3 用法

initialized 对被包装的类型 T 要求很低，仅要求 T 是可缺省构造的，无须可拷贝构造，使用时只需要用 initialized 加上模板参数来声明变量即可。例如：

```
#include <boost/utility/value_init.hpp>
using namespace boost;
int main()
{
  initialized<int> i;            //包装 int 初始化
  assert(i == 0);
  initialized<string> s;         //包装 string 初始化
  assert(s.data().empty());      //注意这里
  initialized<char> d('M');      //传递参数初始化
  assert(d == 'M');
}
```

上面这段代码与 2.4.1 小节的代码类似，但它使用 initialized 的统一语法对变量进行了初始化。

还需要注意的一点是，虽然 initialized 提供了到类型 T 的隐式类型转换，但这必须是在相应的语境中才能实现。在代码中我们无法写出 s.empty()这样的代码，因为 s 的实际类型是 initialized<string>而不是 string，不具有 empty()成员函数。

我们也可以使用自由函数 get()来操作 initialized 对象，它总能获得被包装对象的引用，而且写法更简单，也更通用，就像下面代码所显示的：

```
initialized<string> s;           //初始化字符串 s
assert(get(s).empty());          //使用通用的 get()函数
```

2.4.4 value_initialized<T>

value_initialized 是 Boost1.44 版之后的一个“遗留类”，类摘要如下：

```
template<class T>
class value_initialized
{
public :
  value_initialized() ;
  operator T const &() const ;
```

```
    operator T&();
    T const &data() const ;
    T& data() ;
    void swap( value_initialized<T>& );
  private :
    struct wrapper;
  } ;
```

value_initialized 是 initialized 的“子集”，具有 initialized 的大部分功能，只是没有提供单参构造函数，因此不能直接值初始化。除了这一点，它与 initialized 的功能完全相同，但不推荐使用。

2.4.5 更方便的用法

对于支持拷贝构造的类型，value_initialized 库另外提供了一个类 initialized_value_t 和一个常量实例 initialized_value。

类 initialized_value_t 是一个“空类”，仅有一个到类型 T 的转型操作，用来生成恰当的 initialized 对象，摘要如下：

```
class initialized_value_t
{
public :
  template <class T> operator T() const   //转换到类型 T
  {
    return initialized<T>().data();       //返回初始化好的对象
  }
};
initialized_value_t const initialized_value = {} ;
```

因而我们能够用类似 T v = T() 的语法来更简单地初始化变量，initialized_value 可以这样使用：

```
int i = initialized_value;                //初始化整型变量
assert(i == 0);
string s = initialized_value;             //初始化标准字符串变量
assert(s.empty());
```

如果觉得 initialized_value 的名字过长，不方便书写，我们也可以用 boost.typeof 库来为它更名：

```
BOOST_AUTO(const &v_init, initialized_value);
```

这将声明一个对 initialized_value 的常引用变量 v_init，它是 initialized_value 的一个别名，用法完全相同。

2.5 base_from_member

有时候基类需要由派生类的成员变量来初始化，但通常的写法因 C++的类初始化顺序要求而不能正确实现，因为基类必须在派生类之前完成初始化，而那时派生类的成员是未定义的，解决方法是把派生类的成员移动到另一个辅助基类中。base_from_member 使用多重继承和模板技术提供了这个用成员来初始基类的惯用法。

base_from_member 位于 boost 名字空间，为了使用 base_from_member 组件，需要包含头文件<boost/utility/base_from_member.hpp>[①]，即：

```
#include <boost/utility/base_from_member.hpp>
using namespace boost;
```

2.5.1 类摘要

base_from_member 是一个很简单的类，摘要如下：

```
template < typename MemberType, int UniqueID = 0 >
class base_from_member
{
protected:
    MemberType  member;        //成员变量

    base_from_member();        //构造函数

    template< typename T1 >
    explicit  base_from_member( T1 x1 );

    template< typename T1, typename T2 >
    base_from_member( T1 x1, T2 x2 );

    ...//其他构造函数
```

① 也可以直接包含<boost/utility.hpp>，它内含数个小工具的实现。

```
};
```

base_from_member 有两个模板参数：MemberType 是它的数据成员类型，UniqueID 则是一个起标记作用的整数，用来在使用多个 base_from_member 时进行区分，默认值是 0。

base_from_member 有一个保护数据成员，命名为 member，它的类型就是模板参数中的 MemberType。因为被声明为 protected，所以 member 可以被派生类任意使用，相当于把类的成员用法转化为了继承用法。

默认情况下，base_from_member 有 11 个构造函数，最大支持 10 个参数，这些参数被用来在构造时初始化 member 成员变量。构造函数参数的数量是可以定制的，base_from_member 使用了预处理元编程库 preprocessor，只需要在包含 <boost/utility/base_from_member.hpp>前定义宏 BOOST_BASE_FROM_MEMBER_MAX_ARITY 即可，例如：

```
#define BOOST_BASE_FROM_MEMBER_MAX_ARITY  2   //注意这里
#include <boost/utility/base_from_member.hpp>
```

将使 base_from_member 最多只有三个构造函数，支持最多两个参数。

2.5.2 用法

先来看一下用派生类的成员来初始基类的通常写法，下面的代码不能正确实现编写者的目的：

```
class base                                  //基类
{
public:
  base(complex<int> c)                      //使用标准库的复数初始化
  {
      cout << "base ctor" << endl;
      cout << c << endl;                    //输出复数的值
  }
};                                          //基类定义结束
class derived:public base                   //派生类
{
  complex<int> c;                           //派生类的复数成员
public:
  derived(int a, int b):c(a, b),base(c)     //初始化成员和基类
  {
```

```
        cout << "derived ctor" << endl;
        cout << c << endl;                    //输出复数的值
    }
};                                            //派生类定义结束
int main()
{
    derived d(10, 20);                        //创建一个实例
}
```

程序的运行结果可能是这样：

```
base ctor
(-858993460,-858993460)
derived ctor
(10,20)
```

很明显，基类 base 没有被正确初始化，因为在 base 构造时派生类的成员 c 还没有被初始化，其值是未定义的，如果是在真实的代码中这将导致灾难性的后果。

使用 base_from_member 可以很容易地解决这个问题，只需要把派生类的成员变量声明改为使用 base_from_member 的继承方式，把需要使用成员初始化的基类放在继承列表的最后：

```
class derived:
    private base_from_member<complex<int> >,          //声明成员变量
    public base                     //从基类派生，应在 base_from_member 之后
{
    //complex<int> c;                                 //不直接声明成员变量
    //typedef 简化关于 base_from_member 代码的编写
    typedef base_from_member<complex<int> > pbase_type;
public:
    derived(int a, int b):pbase_type(a, b),base(member)     //初始化
    {   ... }
};
```

派生类 derived 与前一个版本具有少量但关键的变化：

首先，它必须取消原来的成员变量 c 的声明，而改用 base_from_member 来间接声明成员变量。注意在继承时我们使用了 private 而不是 public，这将使 base_from_member 的 member 成员变量在 derived 中成为 private。因为 base_from_member 的名字过长，

所以我们最好定义辅助类型来简化代码，通常这个辅助类型命名为 pbase_type。

这样，在 derived 的构造函数的初始化列表中我们就可以先初始化 base_from_member 定义的辅助基类 pbase_type，因为声明的顺序在前它将先于 base 初始化，从而使成员变量 member 拥有正确的初值，随后基类 base 使用 member 也将被正确初始化。

修改后的代码运行结果如下：

```
base ctor
(10,20)
derived ctor
(10,20)
```

2.5.3 进一步的用法

因为 base_from_member 的成员变量被命名为 member，如果派生类需要使用多个成员变量来初始化基类时就会存在名字冲突，这时必须要使用基类的名字来限定，使用起来有些麻烦。

假设再增加一个基类 base2，它需要使用两个 string 变量初始化：

```
class base2
{
public:
  base2(string &x, string &y)
  {
      cout << "base2 ctor" << endl;
      cout << x <<  y << endl;
  }
};
```

那么如果 derived 要从 base 和 base2 同时继承的话，代码应该如下：

```
class derived:
  private base_from_member<complex<int> >,   //第一个成员
  private base_from_member<string, 1>,       //使用第二个模板参数
  private base_from_member<string, 2>,       //用于区分不同的类型
  public base, public base2                  //最后是真正的基类
{
  typedef base_from_member<complex<int> >    pbase_type;
  typedef base_from_member<string, 1>        pbase_type1;
  typedef base_from_member<string, 2>        pbase_type2;
```

```
public:
  derived(int a, int b):
    pbase_type(a, b),
    pbase_type1("str1"),pbase_type2("str2"),
    base(pbase_type::member),                   //基类名字限定来使用成员变量
    base2(pbase_type1::member, pbase_type2::member)
  {
      cout << "derived ctor" << endl;
      cout << pbase_type::member << endl;
  }
};
```

因为使用了大量的多重继承，这段代码明显比之前的代码要难理解一些，写法也显得有点复杂和冗长，特别是使用 member 成员变量的时候，必须要使用基类名来限定。

因此，我们应当尽量避免使用多于一个的 base_from_member 用法，这将使代码难以理解难以维护。如果确实存在一个需要使用成员变量进行复杂初始化的类，那么最好重新设计类的结构，或者根据 base_from_member 的工作原理自己手工编写一个辅助类。

例如，手工编写辅助类解决上面的派生类问题代码如下所示：

```
class pbase_type                                  //手工编写的辅助类
{
protected:
  complex<int> cp;                                //成员变量都移动到这个辅助类中
  string x,y;
  pbase_type(int a, int b, string c, string d):  //构造初始化
      cp(a,b), x(c), y(d)
  {}
};
class derived: private pbase_type, public base
{
public:
  derived(int a, int b):
    pbase_type(a, b, "str1", "str2"),             //初始化成员
    base(cp)                                      //初始化基类
  {
      cout << "derived ctor" << endl;
      cout << c << endl;
  }
};
```

这个解法比使用 base_from_member 要好看易理解的多。

2.6 conversion

C++是一种静态强类型语言，任何变量都必须有一个类型，不同类型的变量相互操作时必须进行隐式或显式类型转换。隐式类型转换通常由编译器自动完成，而显式类型转换则通常由程序员手工强制完成。

C++提供两种显式类型转换语法：一种是继承自 C 语言的“老”语法，用一对圆括号指明转换的类型；另一种是在 C++98 标准中定义的新式转型操作符，如 static_cast<>、dynamic_cast<>，它们比老式语法更可读，更清晰地表明了转换的意图、不容易出错，而且也很容易使用文本处理工具分析处理。

conversion 库针对 C++98 标准中的转型操作符的缺陷进行了改进，能够更安全、更清晰地进行多态对象间的转型，库中的另一个组件 lexical_cast<>还可以进行字面量的转换（lexical_cast 已经在推荐书目[1]做过介绍）①。

conversion 位于名字空间 boost，为了使用 conversion 组件，需要包含头文件<boost/cast.hpp>，即：

```
#include <boost/cast.hpp>
using namespace boost;
```

2.6.1 标准转型操作符

C++98 标准为显式类型转换定义了四个新的转型操作符：const_cast、static_cast、dynamic_cast 和 reinterpret_cast，它们被用于替换老式的显式类型转换语法，能够避免许多任意转型引起的潜在错误。在学习 conversion 库之前，我们有必要先了解这四个 C++98 标准中的转型操作符②：

- const_cast ：用于增加或者去除 const、volatile 修饰，此外不能执行其他任何转换操作；
- static_cast ：可以显式执行所有编译器可执行的隐式类型转换操作，不能

① conversion 库原来还有一个 numeric_cast 组件，用于数字转换，但现已经被 numberic 库所取代，详见 2.7 小节。不过<boost/cast.hpp>头文件也包含了 numberic 的实现，故仍然可以包含它来使用数字转换功能。

② 提醒读者注意，这些转型操作符不仅可以操作指针，也可以操作引用。

执行多态类的交叉转型；

- dynamic_cast ：用于多态对象（即存在虚函数的对象）间的类型转换，可以向上或者向下转换对象的类型；
- reinterpret_cast：对目标的内存二进制位进行低层次的重新解释。

这四个转型操作符中最常用的是前三个，而第四个 reinterpre_cast 可能对大多数读者都较为陌生。

reinterpre_cast 的作用很接近老式的显式类型转换，可以变更被转换对象的含义。它最常见的应用场景是把一个已经失去类型信息的指针（例如 void*，或者是函数指针）"重新解释"，重新获得正确的类型。例如，下面的代码调用了 Windows API 的 GetProcAddress() 函数加载 DLL 中名为"SocketBind"的接口，将返回的 FARPROC 指针转换为所需的函数指针：

```
typedef int (__stdcall *FuncBind)();                    //函数指针类型定义
FuncBind fbind = reinterpret_cast<FuncBind>             //类型转换
      (GetProcAddress(dllHandle, "SocketBind"));
```

但 reinterpre_cast 应当尽量少用，误用它有时候会得到匪夷所思的结果，例如：

```
int i = 100;
cout << static_cast<double>(i)<< endl;                  //正确
cout << *reinterpret_cast<double*>(&i) << endl;         //错误
```

最后一行代码的输出将是是一个莫名其妙的浮点数-9.25596e+61。

const_cast 和 static_cast 的用法都很简单，下面将重点讨论 dynamic_cast，这也是 conversion 库关注的对象。

2.6.2 多态对象的转型

多态对象间的类型转换可以分为两类：从基类到派生类的向下转型（downcast）和从一个基类到另一个基类的交叉转型（crosscast）。

dynamic_cast 操作符可以同时执行这两种不同含义的转型操作，所以有时候会导致使用混淆。dynamic_cast 的另外一个"缺陷"是转型失败后的行为不一致：对引用转型失败时会抛出异常 bad_cast，而对指针转型失败时则返回一个空指针（NULL）。当然，C++98 标准做出这种设计是有其特定考虑的，因为这可以把转型与测试放在一个表达式中完成，但

很多情况下（特别是泛型编程）会造成一些不大不小的麻烦。

假设有下面的一个多态类继承体系：

```
struct base1                                   //第一个基类
{ virtual ~base1(){};};                        //注意，必须有虚函数
struct base2                                   //第二个基类
{ virtual ~base2(){};};                        //注意，必须有虚函数
struct derived :public base1, public base2     //多重继承
{ virtual ~derived(){};};
```

下面的代码演示了 dynamic_cast 操作符的各种用法：

```
base1 *p = new derived;                        //一个多态对象指针
derived *pd = dynamic_cast<derived*>(p);       //向下转型
base2 *pb2 = dynamic_cast<base2*>(p);          //交叉转型

string *ps = dynamic_cast<string*>(p);         //对指针转型
assert(ps == 0);                               //失败，结果是空指针

try
{
  string& s = dynamic_cast<string&>(*p);       //对引用转型
}
catch (bad_cast& e)
{
  cout << e.what() << endl;;                   //失败，抛出异常
}
```

dynamic_cast 对指针和引用的转型行为不一致很多情况下会引起程序员的困扰，特别是在指针转型时必须编写额外的代码来检查空指针（虽然出现的几率很小），或者使用不太常见的 if 语句临时变量，很容易漏写误写检查代码导致安全隐患。

conversion 库针对 dynamic_cast 提供了增强的转型操作，用两个更安全命名更清晰的多态转型操作 polymorphic_downcast 和 polymorphic_cast 区分了多态对象的向下转型和交叉转型。

2.6.3 polymorphic_downcast

polymorphic_downcast 提供对多态对象指针的向下转型能力，它使用

static_cast 提供高效的转型操作，但不具备 dynamic_cast 的错误检测能力。

polymorphic_downcast 是一个很小的模板函数，实现代码如下：

```
template <class Target, class Source>
inline Target polymorphic_downcast(Source* x )
{
    BOOST_ASSERT( dynamic_cast<Target>(x) == x );     //断言转型成功
    return static_cast<Target>(x);                    //静态转型
}
```

因为 polymorphic_downcast 的转型能力依赖于 static_cast，所以它只能执行向下转型，不能执行其他的转型操作，否则会导致编译错误。例如：

```
base1 *p = new derived;
derived *pd = polymorphic_downcast<derived*>(p);     //正确

base2 *pb2 = polymorphic_downcast<base2*>(p);        //编译错误
string *ps = polymorphic_downcast<string*>(p);       //编译错误
```

polymorphic_downcast 在调试模式下使用 BOOST_ASSERT 宏来断言向下转型必定成功，因此它仅提供有限的（在调试模式下）运行时安全，在发生转型错误时不会抛出异常，使用它时必须由程序员保证转型必定成功，否则会产生未定义行为。例如下面的代码：

```
struct derived2 :public base1
{ virtual ~derived2(){};};
base1 *p = new derived;
derived2 *p2 = polymorphic_downcast<derived2*>(p);   //转型错误
```

这段代码定义了另一个派生类 derived2，然后使用 polymorphic_downcast 把一个指向 derived 对象的指针转型成为了 derived2 对象指针。无论是 debug 还是 release 模式，polymorphic_downcast 都不会报出任何错误。

所以 polymorphic_downcast 实际上是一种“优化”的多态转型操作，使用时必须小心谨慎。

2.6.4 polymorphic_cast

polymorphic_cast 是对 dynamic_cast 的一个简单包装，只能对指针执行向下转

型或者交叉转型操作，它内部替程序员做了空指针检查，如果转型结果是空指针（NULL）则抛出异常。

polymorphic_cast 同样是一个很小的模板函数，实现代码如下：

```
template <class Target, class Source>
inline Target polymorphic_cast(Source* x )
{
    Target tmp = dynamic_cast<Target>(x);         //动态转型
    if ( tmp == 0 ) throw std::bad_cast();        //空指针检查
    return tmp;
}
```

polymorphic_cast 统一了转型失败的行为，对指针转型如果失败，那么它就会抛出 bad_cast 异常，因此我们可以编写 try-catch 块来统一且安全地处理多态对象的转型操作。

polymorphic_cast 的能力基于 dynamic_cast，因此它的用法与 dynamic_cast 完全相同，只需要注意一点：它只能转型指针，不能转型引用。

下面的代码示范了 polymorphic_cast 的用法：

```
base1 *p = new derived;
derived *pd = polymorphic_cast<derived*>(p);          //向下转型
base2 *pb2 = polymorphic_cast<base2*>(p);             //交叉转型
try
{
  derived2 *p2 = polymorphic_cast<derived2*>(p);      //异常
  string *ps = polymorphic_cast<string*>(p);          //异常
}
catch (bad_cast& e)
{
  cout << e.what() << endl;
}
```

大多数情况下我们都可以使用 polymorphic_cast 替代 dynamic_cast，它比 dynamic_cast 命名更清楚，而且更安全易用。

2.6.5 使用模板元编程实现转型

polymorphic_downcast 和 polymorphic_cast 的实现简单明了，也非常易用，但

它们有一个小小的缺陷：只能转型指针而不能转型引用，因此与 C++标准的转型操作符还有一点差距。使用 type_traits 结合 mpl 进行元编程可以弥补这个缺陷。

我们自定义的 my_polymorphic_downcast()函数首先使用 mpl::if_<>元函数移除了 Target 和 Source 的指针或者引用修饰，然后使用静态断言检查类型，完整的实现代码如下：

```
template <class Target, class Source> inline
Target my_polymorphic_downcast(Source x )    //注意，没有*、&操作符
{
  //处理 Target 元数据
  typedef mpl::if_<                          //mpl 选择元函数，类似 if-else
      is_pointer<Target>,                    //判断是否是指针类型
      remove_pointer<Target>::type,          //移除指针运算符
      remove_reference<Target>::type>::type  //移除引用运算符
      T;                                     //获得未经修饰的原始类型
  //处理 Source 元数据
  typedef mpl::if_<                          //处理过程同上
      is_pointer<Source>,
      remove_pointer<Source>::type,
      remove_reference<Source>::type>::type
      S;

  BOOST_STATIC_ASSERT( is_polymorphic<T>::value);    //要求 T 是多态类
  BOOST_STATIC_ASSERT( is_polymorphic<S>::value);    //要求 S 是多态类
  BOOST_STATIC_ASSERT((is_base_of<S, T>::value));    //S 和 T 有继承关系

  return static_cast<Target>(x);
}
```

函数 my_polymorphic_cast()的实现要稍微复杂些，因为 dynamic_cast 对引用和指针的行为不相同，所以我们需要定义一个辅助函数对象 my_polymorphic_cast_imp：

```
template <class Target, class Source, bool isPointer>
struct my_polymorphic_cast_imp;

//模板偏特化转型指针
template <class Target, class Source>
struct my_polymorphic_cast_imp<Target,Source, true>
```

```
{
  Target operator()(Source x )                    //实现类似 polymorphic_cast
  {
      Target tmp = dynamic_cast<Target>(x);
      if ( tmp == 0 ) throw std::bad_cast(); //检查空指针
      return tmp;
  }
};
//模板偏特化转型引用
template <class Target, class Source>
struct my_polymorphic_cast_imp<Target,Source, false>
{
  Target operator()(Source x )
  {
      return dynamic_cast<Target>(x);          //直接调用 dynamic_cast
  }
};
```

接下来的 my_polymorphic_cast() 实现代码与 my_polymorphic_downcast() 基本类似：

```
template <class Target, class Source>
inline Target my_polymorphic_cast(Source x )
{
  typedef mpl::if_<                          //处理 Target 元数据
      is_pointer<Target>,
      remove_pointer<Target>::type,
      remove_reference<Target>::type>::type
      T;
  typedef mpl::if_<                          //处理 Source 元数据
      is_pointer<Source>,
      remove_pointer<Source>::type,
      remove_reference<Source>::type>::type
      S;
  BOOST_STATIC_ASSERT( is_polymorphic<T>::value);
  BOOST_STATIC_ASSERT( is_polymorphic<S>::value);

  return my_polymorphic_cast_imp<Target, Source,
      is_pointer<Target>::value >()(x);                    //元计算是否转型指针
}
```

下面的代码简单测试了这两个自定义转型函数的使用：

```
base1 *p = new derived;
derived *pd = my_polymorphic_downcast<derived*>(p);        //正确
derived &rd = my_polymorphic_downcast<derived&>(*p);

pd = my_polymorphic_cast<derived*>(p);                     //正确
base2 &pb2 = my_polymorphic_cast<base2&>(rd);              //交叉转型
```

2.7 numeric/conversion

数字类型的转换是个看似简单却非常复杂的问题，C++98 标准中定义的许多规则非常微妙，如果源类型的值超过了转型目标类型的范围时就有可能发生未定义行为。numeric/conversion 库中有大量的工具类用于精确处理数字的转换，本书只介绍最简单、结论性的 numeric_cast()，它可以在转型时进行范围检查，如果超出范围就抛出异常 std::bad_cast()的子类。

numeric_cast 位于名字空间 boost::numeric，为了使用 numeric_cast 组件，需要包含头文件<boost/numeric/conversion/cast.hpp>[①]，即：

```
#include <boost/numeric/conversion/cast.hpp>
using namespace boost::numeric;
```

用法

numeric_cast()是一个模板函数，声明如下：

```
template<typename Target, typename Source>
Target numeric_cast( Source arg );
```

numeric_cast 的用法与标准转型操作符很像，但它只能执行对数字类型的转型(is_arithmetic<T>:: value == true)。它的转型行为非常明确：如果数字转型后可以被正确地容纳到目标类型那么它不会有任何问题，否则它就会抛出派生自 std::bad_cast 的异常。

示范 numeric_cast 用法的代码如下：

① numeric_cast<>原本是 conversion 库的一部分，后来被移出并重构形成了这个库，因此出于兼容的目的也可以包含<boost/cast.hpp>。

```
short s = std::numeric_limits<short>::max(); //short 的最大值
int i = numeric_cast<int>(s);                //可安全地转型为 int 值
assert(i == s);

try
{
  char c = numeric_cast<char>(s);            //超过 char 的范围，上溢异常
}
catch (std::bad_cast& e)
{
  cout << e.what() << endl;
}
```

因为 short 类型变量 s 的值为 32767，因此 try-catch 块中把 s 转型为 char 时会抛出 numeric::positive_overflow 异常，表示无法进行数字转换，能够避免很多意想不到的错误。相比之下，标准的转型操作符的行为是不确定的，例如如果把转型改用 static_cast 那么 c 将得到一个莫名其妙的-1 值。

numeric_cast 很容易使用，它可以完全替代 static_cast 进行数字转换，带来更好的安全性。

2.8 pointer

泛化的指针（包括原始指针、智能指针和迭代器）是 C++中一个非常重要的概念，只要一个对象具有 operator*、operator++等类似指针的操作它就可以称为是泛化的指针，能够像指针一样使用。Boost 库为此提供了数个小工具，可以操作泛化的指针，在编写泛型代码时非常有用。

2.8.1 pointee

pointee 是一个很小的元函数，可以推导解引用（operator*）的类型，类似 1.2.6 小节的 remove_pointer<>。它位于名字空间 boost，要使用 pointee，需要包含头文件<boost/pointee.hpp>，即：

```
#include <boost/pointee.hpp>
using namespace boost;
```

类摘要

pointee<>的类摘要如下：

```
template <class P>
struct pointee    //因为使用了元函数转发，所以会有一个内部的 type 类型定义
  : mpl::eval_if<
      detail::is_incrementable<P>,
      detail::iterator_pointee<P>,
      detail::smart_ptr_pointee<P>
    >
{};
```

pointee<>使用了模板元编程技术，接受一个可解引用的泛化指针类型 P，用::type 返回 P 所指向的值类型。

用法

pointee<>不仅能够支持内建指针和标准的迭代器，也支持智能指针——包括标准库的 std::auto_ptr 和 Boost 库的 scoped_ptr、shared_ptr，示范用法的代码如下：

```
//使用元函数 is_same<>比较类型
assert((is_same<pointee<int*>::type, int>::value));
assert((is_same<pointee<auto_ptr<int> >::type, int>::value));
assert((is_same<pointee<string::iterator>::type, char>::value));

// 计算 shared_ptr 的值类型
typedef shared_ptr<int> P;
P p(new int(10));
pointee<P>::type v = *p;
assert(v == 10);
```

注意代码的后面几句，我们使用 pointee<>::type 来获得值类型来赋值，这种情况下也可以直接使用 boost.typeof 库，它会自动推导出赋值表达式的类型：

```
BOOST_AUTO(v, *p);    //等价的赋值操作
```

因为 pointee<>是一个元函数，所以它可以很容易地使用特化来支持其他任意的指针类型：

```
namespace boost
{
```

```
    template <class T>
    struct pointee<some_pointer_type >        //模板特化
    {
        typedef some_define type;             //元计算得到对应的类型
    };
}
```

2.8.2 indirect_reference

indirect_reference 是依赖于 pointee<>的另一个元函数，它的功能与pointee<>类似，但::type 返回的是一个引用类型，相当于 pointee<>::type&。

indirect_reference 位于名字空间 boost，需要包含头文件<boost/indirect_reference.hpp>。

示范 indirect_reference 用法的代码如下：

```
#include <boost/indirect_reference.hpp>
using namespace boost;
int main()
{
  assert((is_same<indirect_reference<int*>::type, int&>::value));
  assert((is_same<indirect_reference<auto_ptr<int> >::type,
                  int&>::value));
  assert((is_same<indirect_reference<string::iterator>::type,
                  char&>::value));
}
```

2.8.3 pointer_to_other

pointer_to_other<>是一个工厂元函数，可以基于一个类型生产出另一个同类型的指针或智能指针，它位于名字空间 boost，需要包含头文件<boost/pointer_to_other.hpp>。

类摘要

pointer_to_other<>使用了模板特化技术，类摘要如下：

```
template<class T, class U>
struct pointer_to_other< T*, U >          //对原始指针特化
{
   typedef U* type;
```

```
};
template<class T, class U, template<class> class Sp>
struct pointer_to_other< Sp<T>, U >        //对智能指针特化
{
   typedef Sp<U> type;
};
```

pointer_to_other<>有数个重载形式，这里仅列出了最常用的两种形式。它有两个模板参数 T 和 U，返回与 T*同样形式但却是指向 U 的指针——正如它的名字，把指针转指向另一个类型。

第一种形式如果 T 是原始指针 T*，那么返回 U*；第二种形式使用了较为罕见的“模板的模板参数”（template template parameters），如果第一个参数是个形如 Sp<T>的智能指针，那么返回一个同样形式的智能指针 Sp<U>。

用法

pointer_to_other<>的应用场景主要是模板元编程，这里给出验证性质的示范代码如下：

```
//第一种形式，返回原始指针
assert((is_same<int*,
  pointer_to_other<void*, int>::type>::value));
assert((is_same<string*,
  pointer_to_other<void*, string>::type>::value));

//第二种形式，返回智能指针
assert((is_same<auto_ptr<int>,
  pointer_to_other<auto_ptr<char>, int>::type>::value));
assert((is_same<scoped_ptr<int>,
  pointer_to_other<scoped_ptr<float>, int>::type>::value));
assert((is_same<shared_ptr<int>,
  pointer_to_other<shared_ptr<string>, int>::type>::value));
```

pointer_to_other<>在我们通常的代码中较少使用，但对于编写泛型代码的库作者来说却是必不可缺的工具，第 6 章的链表节点定义就使用了它，读者可进一步参考。

2.8.4　compare_pointees

比较指针所指向的内容是一个经常会使用的功能，但因为空指针的存在，所以指针内容

的比较要比直接的值比较麻烦很多，Boost 在头文件<boost/utility/compare_pointees.hpp>中提供了两个便利的“指针”比较函数和函数对象，它们是：

- equal_pointees(x,y)：比较两个指针是否相等。如果两个指针都是空指针，那么返回 true；如果只有一个是空指针那么返回 false；否则比较两个指针所指的内容，即*x==*y；
- less_pointees(x,y) ：比较两个指针是否具有小于关系。如果 y 是空指针，那么返回 false；如果 x 是空指针，那么返回 true；否则比较两个指针所指的内容，即*x<*y；
- equal_pointees_t 和 less_pointees_t ：对应上面两个函数的函数对象封装。

两个指针比较函数的声明如下：

```
template<class OptionalPointee>
bool equal_pointees( OptionalPointee const& x, OptionalPointee const& y );
template<class OptionalPointee>
bool less_pointees( OptionalPointee const& x, OptionalPointee const& y );
```

因为 equal_pointees()和 less_pointees()是泛型函数，因此只要参数的行为类似指针就都可以执行比较操作（需支持 operator!和 operator*），“指针”可以是原始指针、智能指针（scoped_ptr、shared_ptr）或者 boost::optional。

示范 equal_pointees()和 less_pointees()用法的代码如下：

```
#include <boost/smart_ptr.hpp>
#include <boost/optional.hpp>
#include <boost/utility/compare_pointees.hpp>
using namespace boost;

int main()
{
  scoped_ptr<int> p1(new int(10));    //两个作用域智能指针
  scoped_ptr<int> p2(new int(20));

  assert(!equal_pointees(p1, p2));    //不相等
  assert(less_pointees(p1, p2));      //小于关系

  p2.reset();                         //p2 变为空指针
  assert(!less_pointees(p1, p2));     //不存在小于关系
```

```
    optional<string> op1, op2;          //两个 optional 对象，均空

    assert(equal_pointees(op1, op2));   //相等
    op2 = "hello";                      //赋值
    assert(less_pointees(op1, op2));    //小于关系
}
```

std::auto_ptr 和迭代器虽然行为也类似指针，但它们不支持 operator!，所以不能应用于这两个函数。

2.9 scope_exit

RAII（资源获取即初始化，Resource Acquisition Is Initialization）是每一个 C++程序员都应该熟悉的技术，这种技术充分利用了构造函数和析构函数会被自动调用的特性，把系统资源的获取和释放动作代码放在构造函数和析构函数中执行，可以确保不发生资源丢失的异常情况，这也是 C++最重要的特性之一。

智能指针如 std::auto_ptr、boost::shared_ptr 是 RAII 的一个很好的应用，它们可以自动地管理原始指针的生命周期，使程序员不必再四处编写 delete 语句来释放指针和指针关联的资源。但智能指针不能解决所有的 RAII 问题，还有很多时候我们必须手工编写 RAII 类来自行管理资源的获取和释放，显得有些麻烦。

scope_exit 库是预处理元编程的一个很好的范例，它使用宏允许直接编写退出作用域时释放资源的代码，很多情况下可以减少编写 RAII 类的工作量，而效果是完全相同的。

为了使用 scope_exit 组件，需要包含头文件<boost/scope_exit.hpp>，即：

```
#include <boost/scope_exit.hpp>
using namespace boost;
```

2.9.1 用法

scope_exit 库提供了两个宏：BOOST_SCOPE_EXIT 和 BOOST_SCOPE_EXIT_END 来实现退出作用域时执行代码，这两个宏必须成对使用。

它们的用法很类似于 try-catch 块或者声明一个函数：以一个 BOOST_SCOPE_EXIT 宏开始，宏的参数是一串捕获变量列表，然后是处理语句块，最后以宏 BOOST_SCOPE_EXIT_END 结束，基本形式如下：

```
BOOST_SCOPE_EXIT(/*变量列表*/) //处理块开始
{
  ...//任意的处理语句
}BOOST_SCOPE_EXIT_END         //处理块结束
```

捕获变量列表可以有多个参数，它们不能使用逗号分隔，必须用圆括号分隔——即使有一个变量也必须用圆括号，这是由预处理元编程库 preprocessor 的特性所决定的，写法上可能稍微有点古怪。

之后的处理语句必须用一对花括号包围起来，同样，即使只有一行代码也必须如此，否则会导致编译错误（其原因会在 2.9.3 小节阐述）。这里的代码可以执行任意的动作——任何合法的、复杂的 C++代码都允许——当然，通常我们应该根据捕获变量的条件编写恰当的资源释放代码，这才是 scope_exit 的本意。

编写完处理语句后，必须用宏 BOOST_SCOPE_EXIT_END 来结束 scope_exit，否则也会导致编译错误。

接下来我们通过几个例子来了解 scope_exit 的用法。

2.9.2 应用示例

首先是一个最简单的例子，它使用 scope_exit 在退出作用域时删除动态分配的内存：

```
#include <boost/scope_exit.hpp>
using namespace boost;

int main()
{
  int *p = new int[100];      //申请一块内存
  BOOST_SCOPE_EXIT((p))       //处理块开始，捕获变量 p
  {
      cout << "scope exit called." << endl;
      delete[] p;             //释放内存
      cout << "scope exit end." << endl;
  }BOOST_SCOPE_EXIT_END       //处理块结束

  cout << "ok. I'm exit." << endl;
}
```

代码非常简单，只是为了显示退出作用域时的执行情况而增加了一些 cout 输出语句，

无论发生什么异常情况，指针 p 分配的内存都会被正确释放。

首先用 new 操作符获取了一块内存，然后立刻用 BOOST_SCOPE_EXIT 宏编写退出时对它的处理语句。之所以要“立刻”编写，是因为如果资源申请与 BOOST_SCOPE_EXIT 块之间如果存在其他语句的话，那么这些语句就有可能发生异常而导致退出作用域，而 BOOST_SCOPE_EXIT 块则因为还没有来得及声明而无法生效。

BOOST_SCOPE_EXIT 的捕获变量列表中直接使用了变量 p，这样 BOOST_SCOPE_EXIT 块内部会使用它的拷贝，对它执行任意操作都不会影响原来的值，这与函数参数声明的效果有些类似。

BOOST_SCOPE_EXIT 块的处理代码很简单，它直接用 delete[]删除指针释放内存。

第二个例子稍微多了点内容，但同样很简单：

```
int main()
{
    bool commit = false                          //标志变量
    string result;
    BOOST_SCOPE_EXIT((&commit)(&result))         //使用引用方式捕获
    {
        if (!commit)                             //检查标志
        {   result = "some error."; }
    }BOOST_SCOPE_EXIT_END                        //处理块结束

    ...                                          //处理 result 的一些操作

    commit = true;                               //提交修改
}
```

这段代码也很简短，定义了一个 bool 变量 commit，并在 BOOST_SCOPE_EXIT 块中检查，实现了类似“commit-rollback”的功能。BOOST_SCOPE_EXIT 的捕获变量列表使用了多个变量，并且都是引用的形式，这样我们就可以直接操作变量自身。

2.9.3 实现原理

scope_exit 看起来似乎很神奇，但它本质上仍然是使用了 RAII 技术，只是通过预处理元编程把实现细节隐藏到了宏的背后而已。

BOOST_SCOPE_EXIT 和 BOOST_SCOPE_EXIT_END 这两个宏在预处理展开后实际上定

义了一个局部 RAII 类和它的实例，我们编写的处理语句成为了它的一个静态成员函数，在类的析构函数被调用。

经过预处理后的 scope_exit 代码如下（省略了一些名字后缀和大部分细节）：

```
struct boost_se_guard_t                  //自动生成的类
{
  boost_se_guard_t (...) {}              //构造函数
  ~boost_se_guard_t(...)                 //析构函数
  {   boost_se_body(...); }              //调用静态成员函数
  static void boost_se_body(... )        //静态成员函数
  {
      ...                                //实际处理语句
  }
} boost_se_guard(...) ;                  //声明类的实例
```

这样，当离开作用域时，类的实例 boost_se_guard 被析构从而导致处理语句被执行。

scope_exit 使用了 boost.typeof 库，把宏的参数推导出类型并作为生成的函数 boost_se_body()的函数参数，因此它依赖于 typeof 库的能力，如果捕获变量列表中需要使用自定义的类型必须预先用 BOOST_TYPEOF_REGISTER_TYPE 宏向 typeof 库注册，详细的做法可见推荐书目[1]。

2.9.4 注意事项

由于编译器的原因，scope_exit 库在使用时有一些需要注意的地方。

对于某些老版本的 GCC 编译器，如果需要在模板函数中使用 scope_exit，那么需要改用 BOOST_SCOPE_EXIT_TPL 来开始 scope_exit 块，否则会编译失败，代码如下所示：

```
template<typename T>
void test(T  t)
{
  BOOST_SCOPE_EXIT_TPL((t))              //在某些老版本gcc上必须使用TPL后缀的宏
  {
      cout << t << endl;
      cout << "scope exit end." << endl;
  }BOOST_SCOPE_EXIT_END
}
```

Windows 上的 MSVC 和 Mac OS X 上的 Xcode4 不存在此问题，如果为了使代码具有

更好的可移植性，在模板函数中最好总使用宏 BOOST_SCOPE_EXIT_TPL。

2.10　总结

本章我们讨论了 Boost 程序库中数个有用的小工具，要求读者对 C++中的缺省构造函数、拷贝构造函数、对象的初始化、操作符重载、类型转换等许多基本概念有较深刻的理解，我们还使用模板元编程概念对其中的一些组件做了深入的剖析和扩展。

compressed_pair 是我们看到的第一个小工具，它是对 std::pair 的增强，对于库作者来说它非常有用，因为可以统一地处理空类和非空类，使空间占用得到尽可能的优化。指针容器库 ptr_container（第 5 章）的 auto_type 类型就使用了它。

checked_delete 是一个“智能操作符 delete”，可以代替 delete，检查删除的对象是否是一个完整类，能够避免很多运行时可能发生的错误，它对应的“智能操作符 new”是 factory 函数对象（4.3 小节）。address_of 是另一个智能操作符，它比内建的 operator&更好，总能够获得真实的对象地址，但因为使用了多次强制类型转换，在运行效率上有一点损失。通常情况下我们不应该重载 operator&，也应该少使用 address_of。

C++11 中对变量的初始化给出了明确的定义，但现在仍然被广泛使用的 C++98 并没有明确定义，所以在泛型初始化变量时我们可以使用 value_initialized，它能够保证对象总被零初始化或者缺省初始化，对于可拷贝构造的类型还可以用更方便的 initialized_value 直接赋值初始化。

base_from_member 对基类使用子类成员初始化提供了一个标准的解决方案，在只有一个成员的时候可以简化相当多的工作，因而被 Boost 库的许多其他组件所使用，例如 iostreams 库（第 8 章）的 stream 类。但在多重继承或基类的初始化比较复杂的时候它就不太适用了，这时我们可以基于它的工作原理实现自己的 base_from_member 辅助类。

类型转换是 C 语言遗留的传统，老式的类型转换风格很差，应当尽量少用，因为它使 C++的静态强类型特性部分地失去了效力，真正好的程序应该少使用。在万不得已的情况下类型转换应该使用新式的转换操作符，Boost 也为此提供了专门针对多态转型 polymorphic_downcast/polymorphic_cast 和数字转型的操作函数 numeric_cast，使用它们能够让代码更加整洁干净和更少错误。关于类型转换的话题讨论还没有结束，serialization 库另外提供一个更智能的转型工具 smart_cast（9.9.3 小节）。

泛化的指针是泛型编程中经常操作的对象，Boost 库有多个小工具专门由于处理指针类型，它们在通常的开发工作中可能用到的机会很少，但学习它们有利于我们理解 Boost 库其他组件的工作原理。

本章最后讨论的是 scope_exit，它基于 RAII 技术使用预处理元编程简化了 RAII 对象的编写工作，能够轻松编写可靠的释放资源代码，在某些时候非常方便有用。

第3章

迭代器

迭代器是现代 C++编程中的重要角色，是连接容器和算法的“粘接剂”，在 STL 中占有非常重要的地位，而 Boost 又对迭代器的演化做出了重要的贡献。

本章首先讨论迭代器设计模式和 C++98 标准中迭代器的基本知识，然后在此基础上讲解 Boost 的新式迭代器定义和分类，之后重点研究 iterators 库里提供的迭代器工具，它们大大简化了程序员编写符合标准的迭代器所需要做的工作。阅读本章需要读者对迭代器模式、适配器模式和标准库的迭代器有一定的了解，必要时请结合推荐书目学习。

Boost 库的迭代器功能并没有归在一个统一的头文件中，而是分散成了许多小的头文件，显得有些凌乱，使用时必须根据具体情况包含特定的头文件。

3.1 迭代器概述

本小节中我们将简要讨论迭代器设计模式，回顾 C++98 中迭代器相关的各种概念和工具，并介绍 Boost 的新式迭代器概念，这些是本章的基础所在。

3.1.1 迭代器模式

推荐书目[2]中对迭代器模式的描述如下：

“提供一种方法顺序访问一个聚合对象中各个元素，而又不暴露该对象的内部表示。”

迭代器模式是一个行为模式，它把聚合的表示与其中元素的访问分离开来，这两者都可以独立地变化，增强了使用的灵活性，因此几乎所有面向对象的系统中都应用了迭代器模式。

迭代器模式有两个基本的参与者：聚合和迭代器。聚合定义了元素的集合方式，它对用户是不透明的，但对迭代器开放了访问接口，允许迭代器访问。迭代器依赖于聚合，它对外提供访问和遍历聚合的接口，这样用户无需关心聚合的内部结构就能够以任意的方式访问聚合里的元素。聚合和迭代器是彼此独立的，这意味着它们都可以是多态的（包括静态多态和动态多态），而且在一个聚合上可以执行多个不同的迭代，一个迭代器也可以同时对多个聚合执行迭代。

除了聚合和迭代器，迭代器模式还可能有其他参与者，例如迭代器产生器，它产生基于某个聚合的某个访问方式的迭代器。迭代器产生器可以是一个单独的工厂类，也可以是聚合或者迭代器的某个工厂方法。

为了访问和遍历聚合，迭代器必须具备四个操作接口：First、Next、IsDone 和 CurrentItem，分别用于执行迭代器初始化、前进、检测是否完成迭代和访问当前元素。基本接口以外迭代器还可以拥有更多的接口以提供更强大更方便的访问和遍历功能，例如比较迭代位置、后退、前进 N 个位置等。

我们可以用 C++标准库中的向量容器 std::vector 和 std::vector::iterator 来具体理解一下迭代器模式：

std::vector 定义了一个元素的聚合，其内部实现对用户来说是不透明的（虽然大多数情况下是一个原生数组），std::vector::iterator 是基于这个聚合的一个迭代器，它可以正向遍历 std::vector。std::vector 的成员函数 begin()/end()是迭代器产生器，它可以产生聚合上的迭代器，同时它们还对应迭代器的 First 和 IsDone 操作，确定了迭代器的起点和终点。std::vector::iterator 重载了 operator++和 operator*，可以在聚合上前进和访问聚合里的元素，对应迭代器的 Next 和 CurrentItem 操作。std::vector::iterator 还重载了 operator==、operator--和 operator+=等操作符，可以在 std::vector 上访问任意位置上的元素。

3.1.2 标准迭代器

C++98 标准中将迭代器分为五类，它们的特征简要描述如下[①]：

- 输入迭代器 ：或称“只读迭代器”（从迭代器输入），只提供 operator++，可以执行相等比较；
- 输出迭代器 ：或称“只写迭代器”（向迭代器输出），只提供 operator++，没

① 读者可参考 10.3.4 小节“迭代器概念检查”对照阅读。

有相等比较功能；

- 前向迭代器　　：可以读写，只提供 operator++，可以执行相等比较和赋值操作；
- 双向迭代器　　：在前向迭代器的基础上增加了 operator--，也就是说可以执行后退操作；
- 随机访问迭代器：在双向迭代器的基础上增加了迭代器的算术运算功能，提供 operator[] 和 operator+=。

对于标准容器来说，std::list、std::set、std::map 的迭代器都是双向迭代器，而内建数组、std::vector、std::deque、std::string 的迭代器则是随机访问迭代器。

为了区分不同类型的迭代器，标准库使用一些 tag 类（空类）作为迭代器的标签，如 std::input_iterator_tag、std::output_iterator_tag，这些类别信息可以使用特征类 std::iterator_traits<>提取（3.3.1 小节）。

标准库的对迭代器的分类是一个重要的成果，但它把迭代器的取值和遍历这两个正交（不相关）的操作混合在了一起，因而被认为存在缺陷[①]，而且造成许多现实中的迭代器无法被恰当的归类。

3.1.3 新式迭代器

iterators 库定义了一组基于 STL 的新的迭代器概念、构造框架和有用的适配器，能够帮助程序员更轻松地应用迭代器模式来创建、使用迭代器类型。这里我们先讨论它的概念定义，库提供的工具将在随后介绍。

iterators 程序库针对标准库的不足区分了迭代器的值访问概念和遍历概念，重新划分了迭代器的类型，使迭代器的概念描述更加清楚[②]。

根据迭代器的值访问概念，迭代器可分为以下四类：

- 可读迭代器　：提供 operator*，可返回可转换为类型 T 的右值；
- 可写迭代器　：提供 operator*，可以执行赋值操作；
- 可交换迭代器：两个迭代器所指的值可用标准库的 iter_swap() 函数交换，即同时满足可读迭代器和可写迭代器的要求。

① C++98 的这个迭代器分类据称可能会在 C++11 标准中取消。
② 这些新式迭代器概念可参考 10.3.5 小节阅读，那里提供了对这些迭代器概念的检查功能。

■ 左值迭代器 ：满足可交换迭代器，并且 operator*可返回左值，即一个类型 T 的引用。

根据迭代器的遍历概念，迭代器可分为以下五类（下面的迭代器概念均在前一个的基础上递增定义）：

■ 可递增迭代器 ：提供 operator++，可拷贝构造和赋值；

■ 单遍迭代器 ：增加 operator==、operator!=比较操作；

■ 前向遍历迭代器 ：增加 difference_type 类型定义，可计算迭代器的距离，可缺省构造；

■ 双向遍历类代器 ：增加 operator--，可以逆向遍历；

■ 随机访问遍历迭代器：增加迭代器的算术运算和比较运算，并提供 operator[]。

使用 Boost 的新式迭代器分类，标准库原有的五个迭代器类别就可以更精确地定义如下：

■ 输入迭代器 =可读迭代器+单遍迭代器；

■ 输出迭代器 =可写迭代器+可递增迭代器；

■ 前向迭代器 =左值迭代器+前向遍历迭代器；

■ 双向迭代器 =左值迭代器+双向遍历迭代器；

■ 随机访问迭代器 =左值迭代器+随机访问遍历迭代器。

而标准库中“不是容器的容器”vector<bool>的迭代器（它返回一个代理而不是 bool）也就可以被正确归类为可交换迭代器（可读迭代器+可写迭代器）+随机访问遍历迭代器。

新式迭代器分类与标准迭代器分类的关系可以用下面的表格描述：

	可读	可写	可交换	左值
可递增		标准输出迭代器		
单遍	标准输入迭代器			
前向				标准前向迭代器
双向				标准双向迭代器
随机			vector<bool>的迭代器	标准随机迭代器

表格中的空白部分是标准迭代器分类所没有覆盖的部分，也就是说存在着这样的新式迭代器可以使用。

3.1.4　标准迭代器工具

为了方便地操作迭代器，标准库提供了若干个辅助工具，它们可以让迭代器用起来更加容易。

迭代器辅助函数

标准库提供了三个迭代器辅助函数：

- advance(pos, n)　：使迭代器前进或后退 n 个位置；
- distance(p1, p2)：计算两个迭代器间的距离；
- iter_swap()　　　：交换两个迭代器所指的元素的值。

这三个迭代器辅助函数可以以统一的方式操作迭代器，而无需关心迭代器的类别，例如，使用 advance()函数，即使是不支持随机访问的迭代器也可以前进或后退任意步，增强了程序的灵活性，3.2 小节的 next_prior 就使用了 advance()。

迭代器适配器

标准库另外提供一些迭代器适配器，可以把一种迭代器转换为另一种迭代器：

- 逆向迭代器：适配后迭代器可以逆序迭代；
- 插入迭代器：把赋值操作转换为执行插入操作的输出迭代器；
- 流迭代器　：把 IO 流转换为迭代器操作。

这三种迭代器适配器中较常用的是后两者，可以让我们以操作迭代器的方式去操作容器和流，例如：

```
using namespace boost::assign;          //打开 assign 名字空间
vector<int> v1, v2;                     //两个标准容器

push_back(v1),1,2,3,4,5;                //使用 assign 库添加数据

std::copy(v1.begin(), v1.end(),         //copy 算法操作迭代器
    back_inserter(v2));                 //使用插入迭代器把容器转换为输出迭代器
assert(v2.size() == 5);
```

```
std::copy(v2.begin(), v2.end(),         //copy 算法操作迭代器
    ostream_iterator<int>(cout));       //使用流迭代器向流输出数据
```

关于这些迭代器辅助工具更多的知识请参考推荐书目[3]。

3.1.5　迭代器与算法

C++98 标准库提供了大量的标准算法，这些算法并不直接操作容器，而是以迭代器指定一个容器的区间，然后通过迭代器来操作容器里的元素。某种程度上来说，算法真正操作的实际上是迭代器，它并不了解容器。

在本章中我们最常用的算法是变动算法 std::copy，它的声明如下：

```
template <class InputIter, class OutputIter>
inline OutputIter copy(InputIter first, InputIter last,
                           OutputIter result);
```

std::copy 顾名思义，它正向遍历[first, last)迭代器区间，通过迭代器把其中的所有元素拷贝到目标区间 result 里，然后返回目标区间的最后位置——通常这个返回值被忽略。

std::copy 还有许多同族的算法，如 copy_backward、reverse_copy、remove_copy 等，读者可自行了解。

3.2　next_prior

next_prior 组件提供两个非常简单的模板函数 next()和 prior()，使用迭代器辅助工具 advance()为仅提供 operator++和 operator--的迭代器增加了前向或者后向 N 步的通用解法。

next_prior 位于名字空间 boost，为了使用 next_prior 组件，需要包含头文件<boost/next_prior.hpp>[①]，即：

```
#include <boost/next_prior.hpp>
using namespace boost;
```

① 也可以包含<boost/utility.hpp>，它含有数个小工具的实现。

3.2.1 函数声明

next()和prior()的实现代码很少，故本书把它们全部摘录如下：

```
template <class T>
inline T next(T x)                    //单参形式
{ return ++x; }
template <class T, class Distance>
inline T next(T x, Distance n)        //双参形式
{
    std::advance(x, n);               //调用 advance()辅助函数
    return x;
}

template <class T>
inline T prior(T x)                   //单参形式
{ return --x; }
template <class T, class Distance>
inline T prior(T x, Distance n)       //双参形式
{
    std::advance(x, -n);              //调用 advance()辅助函数
    return x;
}
```

next()和prior()各有两种重载形式，单参版本只前进或后退一步，双参版本则前进或后退 n 步。它们使用了标准库中的迭代器辅助函数 advance()，会自动根据迭代器的类型采取最有效率的步进措施——对于随机访问迭代器直接调用 operator+=，而其他种类的迭代器则循环调用递增或递减操作。

如果迭代器不提供 operator++或 operator--（例如输入迭代器、输出迭代器和前向迭代器没有 operator--），那么使用这两个函数会引发编译错误。

3.2.2 用法

next()和prior()的代码虽然简单，但它们很好地统一了迭代器的操作方式，因为只有随机访问迭代器才能够编写如 iter + n 这样的代码，不利于泛型编程。有了这两个函数，我们就可以写出对所有迭代器类型一致的操作代码。

另外需要注意的是虽然它们的名字叫 next（后继）和 prior（前驱），但双参版本的第二个参数是可以传入负数的，也就是说它们实际上可以使迭代器任意前后移动（前提是迭代器支持 operator--）。

下面的代码定义了一个模板函数 get_n()，它可以返回迭代器后 n 个位置的值：

```
template<typename I>
typename iterator_traits<I>::value_type      //使用 iterator_traits
get_n(I& iter, int n)                        //引用方式传递迭代器变量
{
  return *(next(iter, n));        //也可以写成 return *(prior(iter, -n));
}

int main()
{
  list<int> l = assign::list_of(1)(2)(3)(4); //assign 库初始化

  assign(get_n(l.begin(), 1)  == 2);
  assert(get_n(l.end(), -1)  == 4);
  assert(get_n(l.end(), -2)  == 3);
}
```

这段代码中我们使用的容器是 std::list，它的迭代器是双向迭代器，不支持随机访问，因此如果要访问任意位置的元素通常需要使用一个循环来迭代或者是 std::advance()。而现在有了 next()就方便多了，泛型函数 get_n()可以用一个简单的语句完成这项工作，比直接调用 std::advance()好看的多。

next()的另一个好处是它是以值的方式使用迭代器，因而不会产生 std::advance()改变迭代器位置的副作用，不必为了保持迭代器原位置不变来声明一个变量临时保存迭代器位置。如果不使用 next()，那么我们必须这么写：

```
template<typename I>
typename iterator_traits<I>::value_type
get_n(I& iter, int n)          //引用方式传递迭代器变量
{
  I tmp = iter;                //为不变动 iter 的位置必须使用一个临时变量
  std::advance(tmp, n);        //tmp 前进 n 个位置，而 iter 不变
  return *tmp;
}
```

本书的第 6 章和第 12 章的部分代码使用了这两个工具函数，读者可做进一步参考。

3.3 iterator_traits

iterator_traits 库提供了标准的元函数来访问迭代器的基本属性，较标准库原有的 std::iterator_traits 而言它更加规范，更容易被用在泛型编程和模板元编程中。

iterator_traits 位于名字空间 boost，为了使用 iterator_traits 组件，需要包含头文件<boost/iterator/iterator_traits.hpp>，即：

```
#include <boost/iterator/iterator_traits.hpp>
using namespace boost;
```

3.3.1 标准迭代器特征类

标准库提供 std::iterator_traits<>来获得迭代器（或指针）的属性，基本实现代码如下：[①]

```
template <class Iterator>                        //针对标准迭代器类型
struct iterator_traits {
  typedef typename Iterator::iterator_category  iterator_category;
  typedef typename Iterator::value_type         value_type;
  typedef typename Iterator::difference_type    difference_type;
  typedef typename Iterator::pointer            pointer;
  typedef typename Iterator::reference          reference;
};
template <class T>                               //针对原生指针类型特化
struct iterator_traits<T*> {
  typedef random_access_iterator_tag            iterator_category;
  typedef T                                     value_type;
  typedef ptrdiff_t                             difference_type;
  typedef T*                                    pointer;
  typedef T&                                    reference;
};
```

iterator_traits<>接受一个迭代器（指针）类型，可以获得迭代器（指针）所必备的五个类型信息：

① 如果迭代器的定义不符合规范，那么 std::iterator_traits<>就不能获得这些信息了。

- iterator_category：迭代器的分类，参见 3.1.2 小节；
- value_type　　　：迭代器所指的值类型；
- reference　　　　：迭代器的值引用类型；
- pointer　　　　　：迭代器的指针类型；
- difference_type　：迭代器的距离类型。

std::iterator_traits<>可以正常工作，而且它已经被使用了很多年，但从模板元编程的角度来看它不符合元函数的规范定义，是非标准元函数，难以在更广泛的领域中使用。

3.3.2 类摘要

iterator_traits 库把 std::iterator_traits<>中被“揉成一团”的元数据分解开来，形成了五个标准元函数，而功能则是完全相同的：

- iterator_category<I>　：返回迭代器的分类；
- iterator_value<I>　　　：返回迭代器所指的值类型；
- iterator_reference<I>：返回迭代器的值引用类型；
- iterator_pointer<I>　　：返回迭代器的指针类型；
- iterator_difference<I>：返回迭代器的距离类型。

实际上，这五个元函数没有做任何自己的工作，只是调用了 std::iterator_traits<>，把非标准的内部类型定义转化成了标准的::type 返回。

例如，iterator_value<>的实现代码如下：

```
// boost::detail 名字空间里的 iterator_traits 元函数定义
template <class Iterator>
struct iterator_traits              //元函数转发给 std::iterator_traits<>
    : std::iterator_traits<Iterator>
{};

//iterator_value<>的实现
template <class Iterator>
struct iterator_value               //调用 boost::detail:: iterator_traits<>
```

```
{
    typedef typename iterator_traits<Iterator>::value_type type;
};
```

3.3.3 用法

因为 iterator_traits 库提供的这五个元函数符合标准定义，所以它们用起来要比 std::iterator_traits<>更加灵活方便，可以很容易地与其他元编程工具混合使用，发挥更大的作用。

示范这五个元函数用法的代码如下：

```
typedef int* I;         //一个指针类型的迭代器
assert((is_same<iterator_value<I>::type, int>::value));
assert((is_same<iterator_reference<I>::type, int&>::value));
assert((is_same<iterator_category<I>::type,
              std::random_access_iterator_tag>::value));

//标准容器 list 的 const 迭代器
typedef list<int>::const_iterator II;
assert((is_same<iterator_value<II>::type, int>::value));
assert((is_same<iterator_reference<II>::type,
               const int&>::value));
assert((is_same<iterator_category<II>::type,
              std::bidirectional_iterator_tag>::value));
```

3.4 iterator_facade

iterator_facade（迭代器外观）是 iterators 库的一个重要组件，它使用外观模式提供一个辅助类，能够帮助程序员更容易地创建符合标准的迭代器，比标准库的 std::iterator<>更容易使用，而且更不容易犯错误。iterator_facade 定义了数个迭代器的核心接口，用户只需要实现这些核心功能就可以编写正确且完备的迭代器。

iterator_facade 位于名字空间 boost，为了使用 iterator_facade 组件，需要包含头文件<boost/iterator/iterator_facade.hpp>，即：

```
#include <boost/iterator/iterator_facade.hpp>
using namespace boost;
```

3.4.1 迭代器的核心操作

一个完备的迭代器拥有相当多的外部接口和内部的类型定义，但它存在一个必需的核心操作集合，这个集合定义了迭代器的必需功能。

iterator_facade 要求用户的迭代器类必须实现下面的五个功能（共六个接口，但依据迭代器的类型某些可以不实现）：

- 解引用 ：dereference() const，实现可读迭代器和可写迭代器必需；
- 相等比较：equal() const，实现单遍迭代器必需；
- 递增 ：increment()，实现可递增迭代器和前向遍历迭代器必需；
- 递减 ：decrement()，实现双向遍历迭代器必需；
- 距离计算：advance() 和 distance_to() const，实现随机访问遍历迭代器必需。

这些核心操作将被 iterator_facade 用来实现迭代器的外部接口，所以这些函数通常应该是 private 的[①]。为了使 iterator_facade 能够访问这些核心操作函数，库又提供了一个辅助类 iterator_core_access，它定义了若干静态成员函数可以访问 private 核心操作，用户迭代器需要把它声明为友元。

此外，因为迭代器有时需要拷贝和赋值，故用户自定义迭代器通常还需要有拷贝构造函数和缺省构造函数。

3.4.2 类摘要

iterator_facade 基于迭代器核心操作实现迭代器功能，类摘要如下：

```
template <
    class Derived,                          //迭代器子类
    class Value,                            //迭代器值类型
    class CategoryOrTraversal,              //迭代器的分类
    class Reference  = Value&,              //迭代器的值引用类型
    class Difference = ptrdiff_t,           //迭代器的距离类型
>
```

① 当然，也可以把这些核心操作直接声明成 public（generator_iterator）就是如此，但一般情况下不应该这么做。

```
class iterator_facade {
 public:
    //迭代器各种必需的类型定义
    typedef      remove_const<Value>::type value_type;
    typedef      Reference reference;
    typedef      Value* pointer;
    typedef      Difference difference_type;
    typedef      some_define iterator_category;

    //迭代器的各种操作定义
    Reference    operator*() const;
    some_define operator->() const;
    some_define operator[](difference_type n) const;
    Derived&     operator++();
    Derived      operator++(int);
    Derived&     operator--();
    Derived      operator--(int);
    Derived&     operator+=(difference_type n);
    Derived&     operator-=(difference_type n);
    Derived      operator-(difference_type n) const;
};
... //许多关系运算符定义
```

iterator_facade 基于用户迭代器的六个核心操作实现了数个迭代器接口，包括 operator++、operator--、operator*以及关系运算符。它有五个模板参数，但后两个可以使用缺省参数，我们通常只需要关心前三个。

第一个模板参数 Derived 是子类的名字，也就是用户正在编写的自己的迭代器（即基类链技术①）。第二个模板参数 Value 是迭代器的值类型，第三个模板参数 CategoryOrTraversal 定义了迭代器的分类，它即可以使用标准分类也可以使用新式分类（参见 3.1 小节）。

由于 CategoryOrTraversal 的使用比较复杂，下面把它的取值单独列出：

- std::input_iterator_tag ：标准的输入迭代器；
- std::output_iterator_tag ：标准的输出迭代器；
- std::forward_iterator_tag ：标准的前向迭代器；

① 又称 Curiously Recurring Template Pattern (CRTP)，读者可参见推荐书目[1]对 operators 库的论述。

- std::bidirectional_iterator_tag ：标准的双向迭代器；
- std::random_access_iterator_tag ：标准的随机访问迭代器；
- boost::incrementable_traversal_tag：可递增遍历迭代器；
- boost::single_pass_traversal_tag ：单遍迭代器；
- boost::forward_traversal_tag ：前向遍历迭代器；
- boost::bidirectional_traversal_tag：双向遍历迭代器；
- boost::random_access_traversal_tag：随机访问遍历迭代器。

3.4.3 用法

使用 iterator_facade 之前我们必须要规划自己的迭代器的几个基本特征，包括迭代器的名称、在何种集合上迭代、迭代的对象（解引用的类型）和迭代遍历的类型。

首先，迭代器要用 public 的方式继承 iterator_facade<>，同时指定 iterator_facade 的模板参数，第一个模板参数必须是迭代器自己（用于基类链），然后是值类型和迭代器分类，最后两个模板参数可以使用缺省值。

接下来我们要考虑迭代器的构造函数：迭代器必须能够拷贝构造和赋值（可递增迭代器概念），如果是前向遍历迭代器则还需要缺省构造函数。构造函数通常需要传入被迭代的集合对象。还要有一个成员变量来保存迭代的位置，这样我们才能执行递增或递减操作。

最后我们就可以实现迭代器所必须的核心操作函数了，这些核心函数最好是 private 的，并声明友元类 iterator_core_access 授予 iterator_facade 的访问权。

下面我们使用两个简单的例子示范 iterator_facade 的用法，更多的例子可见 3.5 小节和 3.6 小节。

示例 1

作为第一个示例，我们定义一个在 std::vector 上的可写单遍迭代器 vs_iterator①，所以需要使用 boost::single_pass_traversal_tag 作为迭代器分类标志。因为这个迭代器是可写而不是只读的，不是输入迭代器，所以不能使用 std::input_iterator_tag（当然用了也不算大错）：

① 读者可参考 10.3.5 小节的新式迭代器概念检查验证 vs_iterator 的迭代器属性。

```
template<typename T>
class vs_iterator :
        public boost::iterator_facade<          //基类链技术继承
        vs_iterator<T>, T,                      //子类名和值类型
        boost::single_pass_traversal_tag>       //单遍迭代器类型
```

然后实现迭代器的内部迭代位置存储、构造函数和赋值函数：

```
{
private:
  std::vector<T> &v;                                //容器的引用
  size_t current_pos;                               //迭代器的当前位置
public:
  vs_iterator(vector<T> &_v, size_t pos = 0):      //构造函数
    v(_v), current_pos(pos)
  {}
  vs_iterator(vs_iterator<T> const& other):        //拷贝构造函数
      v(other.v), current_pos(other.current_pos)
  {}
  void operator=(vs_iterator<T> const& other)      //赋值函数
  {
      this->v = other.v;
      this->current_pos = other.current_pos;
  }
```

因为 vs_iterator 是单遍迭代器，没有太多的功能，所以不需要实现所有的迭代器核心操作，只要解引用、递增和比较就足够了：

```
private:
  friend class boost::iterator_core_access;        //必须的友元声明

  reference dereference() const                    //解引用操作
  {   return v[current_pos]; }

  void increment()                                 //递增操作
  {   ++current_pos;  }

  bool equal(vs_iterator<T> const& other) const    //比较操作
  {   return this->current_pos == other.current_pos;}
};
```

这样，只用了三十多行代码，我们就轻松完成了一个完全符合标准的迭代器类型。vs_iterator 用起来和容器内置的迭代器差不多，示范 vs_iterator 用法的代码如下：

```
int main()
{
  using namespace boost::assign;
  vector<int> v = (list_of(1),2,3,4,5);          //一个标准容器

  vs_iterator<int> vsi(v), vsi_end(v, v.size()); //声明两个迭代器
  *vsi = 9;                              //使用迭代器的 operator*操作集合里的元素

  std::copy(vsi, vsi_end,                        //copy 算法
      ostream_iterator<int>(cout, ","));         //使用流迭代器输出到标准流
}
```

程序运行结果如下：

```
9,2,3,4,5,
```

如果把 vs_iterator 模板参数中的值类型改为 const T，那么 vs_iterator 就会变成一个标准的输入迭代器（可读不可写迭代器），下面的语句将无法通过编译：

```
*vsi = 9;                                        //编译错误
```

很显然，如果使用标准的 std::input_iterator_tag 分类标志无法精确描述可写单遍迭代器的特征。

示例 2

接下来我们定义一个每次跳跃式前进 N 个位置的步进迭代器 step_iterator[①]，它接受一个迭代器 I 和整数 N 作为模板参数，递增时调用 N 次 operator++。

首先迭代器还是要选择恰当的，继承自 iterator_facade<>的参数：

```
template<typename I, std::ptrdiff_t N = 2>          //缺省一次前进 2 步
class step_iterator:
  public boost::iterator_facade<
      step_iterator<I>,                             //子类名
      typename boost::iterator_value<I>::type const, //元函数获得值类型
      boost::single_pass_traversal_tag>             //单遍分类
```

① 为了使代码简单起见，这里没有对迭代器是否越界的检查，请读者注意。

然后是迭代器位置存储、构造函数和赋值函数：

```
{
private:
  I m_iter;                                        //迭代器位置
public:
  step_iterator(I x):                              //构造函数
      m_iter(x)   {}

  step_iterator(step_iterator const& other):       //拷贝构造函数
  m_iter(other.m_iter){}

  void operator = (step_iterator const& other)     //赋值函数
  {     m_iter = other.m_iter;  }
```

同样我们需实现单遍迭代器必需的解引用、递增和比较操作：

```
private:
  friend class boost::iterator_core_access;     //必需的友元声明

  reference dereference() const                 //解引用操作
  {   return *m_iter; }

  void increment()
  {   std::advance(m_iter, N);    }             //连续前进N次,不能用m_iter+=N

  bool equal(step_iterator const& other) const  //比较操作
  {   return m_iter == other.m_iter;}
};
```

注意在递增操作 increment() 中不能使用 m_iter += N 的形式，因为我们不能假定迭代器是随机访问迭代器，只有随机访问迭代器才能使用 operator+=。另外我们也不能使用 next() 函数（3.2 小节），因为它不改变迭代器的位置，不满足这里的要求。

示范 step_iterator 用法的代码如下：

```
int main()
{
  char s[] = "12345678";                      //字符数组
  std::copy(s, s + 8,                         //copy 算法，原始指针用做迭代器
      std::ostream_iterator<char>(cout));     //流迭代器输出
```

```
    cout << endl;

    //用 char*迭代，使用默认步长 2
    step_iterator<char*> first(s), last(s + 8);
    std::copy(first, last,                      //copy 算法
        std::ostream_iterator<char>(cout));     //流迭代器输出
}
```

程序的运行结果是：

```
12345678
1357        //跳过了偶数位置的元素
```

step_iterator 的另两个使用例子可见 3.6.8 小节和 3.6.11 小节。

3.5 iterator_adaptor

迭代器是一种很容易使用的对象，标准库的很多算法都可以操作迭代器，所以很多时候我们希望把某些对象模拟成一个迭代器的形式来操作，这样能够简化我们的代码。还有的时候程序里已经存在了可用的迭代器，例如数组指针、标准容器的迭代器等，但它们不能满足特定的要求，不方便使用。这两种情况下 iterators 库的 iterator_adaptor 就可以发挥作用，它基于对象适配器模式，主要功能就是把已经存在的类型适配为一个新的迭代器。

iterator_adaptor 位于名字空间 boost，为了使用 iterator_adaptor 组件，需要包含头文件<boost/iterator/iterator_adaptor.hpp>，即：

```
#include <boost/iterator/iterator_adaptor.hpp>
using namespace boost;
```

3.5.1 类摘要

iterator_adaptor 派生自 iterator_facade，同样使用了基类链技术，类摘要如下[①]：

① iterator_adaptor 沿用了 C++标准库中的术语“Base”，这个 Base 与继承毫无关系，实际上相当于适配器模式中的角色 Adaptee，即被适配的对象。为了叙述清晰起见下文中将所有的 Base 都改称 Adaptee，请读者阅读时留意。

```
template <
    class Derived,                              //迭代器子类名
    class Adaptee,                              //被适配的迭代器
    class Value              = use_default,     //值类型
    class CategoryOrTraversal = use_default,    //迭代器分类标志
    class Reference          = use_default,     //迭代器值引用类型
    class Difference = use_default              //迭代器距离类型
>
class iterator_adaptor  : public iterator_facade<Derived,...>
{
    friend class iterator_core_access;          //必需的友元声明
 public:
    iterator_adaptor();
    explicit iterator_adaptor(Adaptee const& iter);
    typedef Adaptee base_type;                  //被适配的原始迭代器类型定义
    Adaptee const& base() const;
private:
    // 适配 Adaptee 实现 iterator_facade 必需的 6 个核心迭代器接口
    reference            dereference() const;
    bool                 equal(iterator_adaptor<> const& x) const;
    void                 increment();
    void                 decrement();
    void                 advance(typename difference_type n);
    difference_type      distance_to(iterator_adaptor<> const& y) const;
protected:
    typedef some_define  iterator_adaptor_;     //自身的类型定义
    Adaptee const&       base_reference() const;
    Adaptee&             base_reference();
   private:
      Adaptee m_iterator;
};
```

iterator_adaptor 有六个模板参数，但通常我们只需要使用前两个。第一个模板参数 Derived 的含义同 iterator_facade，也是自定义的迭代器类型，即子类；第二个模板参数 Adaptee 是要被适配的、已经存在的类型（可以是已经存在的迭代器，或者是任何其他类型），其他的模板参数可以使用缺省值 boost::use_default 自动推导。

iterator_adaptor 是对 iterator_facade 的一个具体实现，因此它使用 iterator_core_access 作为友元，并借用 Adaptee 实现了 dereference()、increment() 等六个核心操作。

iterator_adaptor 有两个构造函数，有参数的构造函数要求传入被适配的 Adaptee 实例，被保存在成员变量 m_iterator。为了方便子类使用，iterator_adaptor 提供 base_reference() 和 base()，它们可以直接获得被适配的原始迭代器对象 m_iterator。

iterator_adaptor 还有一个类型定义 iterator_adaptor_，它就是 iterator_adaptor<Derived,...>自身，这是为了便于子类引用而不必写出一长串的模板参数列表，在写子类代码时可以直接使用。

3.5.2 用法

iterator_adaptor 对适配的对象有一定的要求：Adaptee 必须是可拷贝构造和可赋值的，但不一定要有 operator++，也就是说不一定是一个迭代器。

iterator_adaptor 是 iterator_facade 的子类，故它的用法与 iterator_facade 也比较相似。不过因为我们要适配已经存在了的类型，所以通常只编写需要行为变化的少数核心操作函数就可以了，其他的操作已经由 iterator_adaptor 替我们实现了。

下面的代码把普通数组指针适配为迭代器接口，因为指针本身已经具有了迭代器的操作，所以适配代码相当的简单：

```
template<typename P>                                //适配任意的指针类型
class array_iter:
  public boost::iterator_adaptor<array_iter<P>, P > //适配类型 P
  {
  //type_traits 静态断言保证 P 必须是指针
   BOOST_STATIC_ASSERT(is_pointer<P>::value);
  public:
    array_iter(P x):iterator_adaptor_(x)            //必要的构造函数①
  {}
};
```

array_iter 的使用也非常的简单，可以从一个原生指针很轻松地获得完全的迭代器能力：

```
int main()
{
  int a[10] = {1,2,3};                          //一个整数数组
```

① 在 Xcode4 中这行代码必须写成 array_iter::iterator_adaptor_(x)，否则编译器会找不到 iterator_adaptor_ 的类型定义。

```
  array_iter<int*> start(a), finish(a + 10);     //两个起点和终点迭代器
  start += 1;                                    //起点迭代器前进一个位置

  std::copy(start, finish,                       //copy 算法
      ostream_iterator<int>(cout, ","));         //流迭代器输出，用逗号分隔
}                                                //输出 2,3,0,0,0,0,0,0,0,
```

改变 iterator_adaptor<>的模板参数就可以变动适配后迭代器的能力，充分展现了 iterator_adaptor 强大而灵活的功能，例如下面的迭代器变成了前向迭代器，不能执行迭代器的算术运算：

```
template<typename P>
class array_iter:
  public boost::iterator_adaptor
      <array_iter<P>, P ,
      boost::use_default,                        //值类型使用缺省推导
      boost::forward_traversal_tag>              //变更迭代器的遍历类型
{
public:
  array_iter(base_type x):iterator_adaptor_(x)   //构造函数同前
  {}
};
```

我们将在接下来的 3.6 节中看到更多的 iterator_adaptor 应用例子。

3.6　迭代器工具

在 iterator_facade 和 iterator_adaptor 这两个强大的辅助类的帮助下，iterators 库提供了很多个非常有用的迭代器，它们既是定义良好的迭代器工具，也是逻辑清晰的代码范例，值得我们仔细研究来深入理解迭代器概念。

这些迭代器工具大部分位于<boost/iterator/>目录，少量位于<boost/>目录，并且都提供形如 make_xxx_iterator()的工厂函数方便使用。

3.6.1　共享容器迭代器

共享容器迭代器（shared_container_iterator）把一个被 shared_ptr 管理的容器适配成迭代器的形式来操作，比直接用 shared_ptr 更加简单方便，它位于头文件

<boost/shared_container_iterator.hpp>。

类摘要

shared_container_iterator 的类摘要如下：

```
template <typename Container>
class shared_container_iterator :
    public iterator_adaptor<                        //使用适配器 iterator_adaptor
      shared_container_iterator<Container>,         //迭代器名称
      typename Container::iterator>                 //被适配的容器迭代器
{
  typedef typename Container::iterator iterator_t;
  typedef boost::shared_ptr<Container> container_ref_t;

  container_ref_t container_ref;
public:
  shared_container_iterator() { }

  shared_container_iterator(iterator_t const& x,container_ref_t const& c) :
    super_t(x), container_ref(c) { }
};
```

shared_container_iterator 的代码非常简单，和我们刚刚实现的 array_iter 非常相似，几乎没有什么自己的功能代码，仅仅是一个简单的适配，它的构造函数要求传入容器的引用和容器的迭代器。

用法

shared_container_iterator 非常适合于操作那些使用 shared_ptr 管理的容器，这些容器可以被安全地共享使用，shared_container_iterator “屏蔽”了对 shared_ptr 的操作，使共享容器用起来更像是标准容器。

下面的代码示范了 shared_container_iterator 的用法：

```
#include <boost/shared_container_iterator.hpp>
#include <boost/make_shared.hpp>
using namespace boost;
int main()
{
```

```
    BOOST_AUTO(sv, make_shared<vector<int> >(10));      //共享容器

    //typedef 简化迭代器的定义
    typedef shared_container_iterator<vector<int> > sci_t;

    sci_t first(sv->begin(), sv);                       //迭代器起点
    sci_t last(sv->end(), sv);                          //迭代器终点

    std::fill(first, last, 9);                          //调用 std::fill 算法
}
```

辅助函数

为了方便使用，shared_container_iterator 还提供一个辅助函数 make_shared_container_iterator()，它可以直接产生 shared_container_ iterator：

```
template <typename Container>
shared_container_iterator<Container>
make_shared_container_iterator(typename Container::iterator iter,
                   boost::shared_ptr<Container> const& container) {
  typedef shared_container_iterator<Container> iterator;
  return iterator(iter,container);
}
```

适当地使用 make_shared_container_iterator()，可以不必写出临时变量简化代码，例如：

```
//直接创建共享容器迭代器，无需 typedef 和临时的迭代器变量
std::fill(
  make_shared_container_iterator(sv->begin(), sv),
  make_shared_container_iterator(sv->end(),sv),
  9);
```

另外一个辅助函数 make_shared_container_range()以 pair 的形式返回共享容器的两个端点，符合 boost.range 的概念，可以传递给使用 range 概念的算法，它的声明如下：

```
template <typename Container>
std::pair<                      //返回一个迭代器的 pair，即 range
  shared_container_iterator<Container>,
  shared_container_iterator<Container> >
```

```
make_shared_container_range(boost::shared_ptr<Container> const& container) {
  return
    std::make_pair(
      make_shared_container_iterator(container->begin(),container),
      make_shared_container_iterator(container->end(),container));
}
```

3.6.2　发生器迭代器

发生器迭代器（Generator Iterator）可以把一个函数或者函数对象适配成**输入迭代器**，每次调用 operator++并解引用迭代器都会获得一个值，它位于头文件<boost/generator_iterator.hpp>。

类摘要

generator_iterator 的类摘要如下：

```
template<class Generator>
class generator_iterator
  : public iterator_facade<                       //使用 iterator_facade
        generator_iterator<Generator>,
        typename Generator::result_type,          //值类型
        single_pass_traversal_tag,                //单遍迭代器
        typename Generator::result_type const&    //常引用，可读不可写
    >
{
 public:
    generator_iterator() {}                       //构造函数
    generator_iterator(Generator* g) :            //构造函数
       m_g(g), m_value((*m_g)()) {}

    void increment()                              //递增操作
    {
        m_value = (*m_g)();
    }
    const typename Generator::result_type&
    dereference() const                           //解引用操作
    {
```

```
        return m_value;
    }
    bool equal(generator_iterator const& y) const//相等操作
    {
        return this->m_g == y.m_g && this->m_value == y.m_value;
    }
  private:
    Generator* m_g;                              //函数对象指针
    typename Generator::result_type m_value;     //产生的值
};
```

generator_iterator 的实现代码相当简单，从它的 iterator_facade<>模板参数可以看出它符合输入迭代器的定义（可读不可写的单遍迭代器），也仅实现了所需的最小迭代器核心操作。读者需要注意 generator_iterator 的构造函数，它要求传入一个发生器指针，而不是引用。

用法

generator_iterator 提供一个辅助函数 make_generator_iterator()，用于直接创建发生器迭代器，它的声明如下：

```
template <class Generator>
inline generator_iterator<Generator>
make_generator_iterator(Generator & gen)
{
  typedef generator_iterator<Generator> result_t;
  return result_t(&gen);
}
```

它的接口与 generator_iterator 的构造函数不同，不是指针而是引用，因此用起来更加方便，配合 boost.typeof 库更可以进一步简化创建迭代器的类型声明。

下面的代码使用 generator_iterator 把随机数发生器适配成了迭代器的形式，可以很容易地使用迭代器的操作方式获得随机数：

```
#include <boost/generator_iterator.hpp>
#include <boost/random.hpp>
using namespace boost;
int main()
{
```

```
    rand48 rng;                                      //rand48 随机数发生器
    BOOST_AUTO(iter, make_generator_iterator(rng));  //创建发生器迭代器
    for (int i = 0;i < 5; ++i)
    {
        cout << *++iter << ",";                      //输出 5 个随机数
    }
}
```

注意在这里我们不能使用 std::copy 算法，因为 copy 算法需要知道迭代器的起点和终点，而这个随机数发生器迭代器的结束位置无法确定，如果要使用 copy 算法可参考 3.6.5 小节使用计数迭代器或者 3.6.6 小节的函数输入迭代器。

3.6.3　逆向迭代器

reverse_iterator 把一个迭代器适配成可以逆序遍历的逆向迭代器，修正了 C++98 中逆向迭代器适配器 std::reverse_iterator 的一些缺点，更加好用，它位于头文件 <boost/iterator/reverse_iterator.hpp>。

类摘要

reverse_iterator 可以把原迭代器的前进后退操作转换为反向操作，类摘要如下：

```
template <class Iterator>
class reverse_iterator
    :public iterator_adaptor< reverse_iterator<Iterator>, Iterator >
{
public:
    reverse_iterator() {}
    explicit reverse_iterator(Iterator x)
        : super_t(x) {}
private:
    typename super_t::reference dereference() const    //解引用
    { return *boost::prior(this->base()); }

     void increment() { --this->base_reference(); }     //递增操作
     void decrement() { ++this->base_reference(); }     //递减操作

     void advance(typename super_t::difference_type n) //前进操作
    { this->base_reference() += -n;      }
};
```

因为 reverse_iterator 逆序迭代，因此要求适配的迭代器必须满足双向迭代器概念，即提供 operator--。

用法

reverse_iterator 提供函数 make_reverse_iterator()，可以轻松创建逆向迭代器，下面的代码把原始指针适配成了逆向迭代器，然后逆序打印输出：

```
#include <boost/iterator/reverse_iterator.hpp>
using namespace boost;
int main()
{
    char s[] = "hello iterator.";                     //字符数组

    std::copy(                                         //copy 算法
        make_reverse_iterator(s + sizeof(s) - 1),      //逆序迭代器起点
        make_reverse_iterator(s),                      //逆序迭代器终点
        ostream_iterator<char>(cout));                 //流迭代器输出
}
```

程序的运行结果如下：

```
.rotareti olleh
```

3.6.4 间接迭代器

indirect_iterator 把一个迭代器进行适配，在执行 operator*时再多执行一次解引用操作（即再执行一次 operator*），适合用于查看保存指针、智能指针或者迭代器的容器。它位于头文件<boost/iterator/indirect_iterator.hpp>。

类摘要

indirect_iterator 的类摘要如下：

```
template <
    class Iterator,                //被适配的迭代器
    class Value = use_default,
    class CategoryOrTraversal = use_default,
    class Reference = use_default,
    class Difference = use_default >
```

```
class indirect_iterator
{
 public:
    indirect_iterator();                //构造函数
    indirect_iterator(Iterator x);      //构造函数

    Iterator const& base() const;
    reference operator*() const;
    indirect_iterator& operator++();
    indirect_iterator& operator--();
};
```

用法

indirect_iterator 用法很简单，同样提供工厂函数 make_indirect_iterator()，只需要注意它只能用于元素是“指针”的容器：

```
#include <boost/iterator/indirect_iterator.hpp>
using namespace boost;
int main()
{
   using namespace boost::assign;
   vector<int*> v = (list_of(new int(1)), new int(2));    //指针容器
   //不使用间接迭代器访问元素
   for (BOOST_AUTO(pos, v.begin()); pos != v.end(); ++pos)
   {
       cout << **pos << ",";                              //需用两次解引用
   }
   cout << endl;

   //使用间接迭代器访问元素
   BOOST_AUTO(start,  make_indirect_iterator(v.begin()));
   BOOST_AUTO(finish, make_indirect_iterator(v.end()));
   while(start != finish)
   {
       cout << *start++ << ",";                           //只用一次解引用
   }

   //需及时删除指针避免内存泄露
```

```
    for_each(v.begin(), v.end(), check_deleter<int>());
}
```

间接迭代器带来的好处很明显，它使我们对指针所指元素的操作透明化了，对指针容器操作起来就像是一个普通容器。

读者可对比一下 3.6.1 小节的共享容器迭代器：indirect_iterator 处理对象是存储在容器中的指针或智能指针，而 shared_container_iterator 处理对象是存储在智能指针中的容器，两者都“屏蔽”了中间的一层指针操作。

3.6.5 计数迭代器

counting_iterator 把一个**可递增**的类型适配成迭代器，它位于头文件 <boost/iterator/counting_iterator.hpp>。

类摘要

counting_iterator 使用了元编程来计算迭代器类型等模板参数，简化的类摘要如下：

```
template <
    class Incrementable,              //可递增类型
    class CategoryOrTraversal = use_default,
    class Difference = use_default>
class counting_iterator:
    public iterator_adaptor<
            counting_iterator<... >   //计数迭代器自身
          , Incrementable             //被适配的可递增类型
          , Incrementable const       //值类型
          , traversal                 //迭代器分类
          , Incrementable const&      //值引用类型
          , difference >              //迭代器距离类型
{
public:
     counting_iterator();
     counting_iterator(counting_iterator const& rhs);
     explicit counting_iterator(Incrementable x);

     Incrementable const& base() const;
     reference operator*() const;
     counting_iterator& operator++();
```

```
    counting_iterator& operator--();
private:
    Incrementable m_inc;
};
```

counting_iterator 最重要的模板参数是 Incrementable，对它的要求是可拷贝构造和可赋值的。Incrementable 必须能够执行 operator++操作，如果 counting_iterator 是单遍迭代器、双向遍历迭代器或者随机访问遍历迭代器，则 Incrementable 还应具备 operator==、operator--和算术运算的能力。

Incrementable 通常是整数，但也可以是任何符合以上要求的类型（例如迭代器）。适配后 counting_iterator 把递增递减操作转交给 private 成员 m_inc，解引用时返回 m_inc 的值——也就是说，counting_iterator 为原类型增加了一个 operator*解引用操作，其他的操作均不变。

用法

counting_iterator 可以为一个类型增加解引用操作，把它变得像是一个迭代器，对于某些需要连续增加的类型来说很有用，通过 counting_iterator 适配后能够搭配标准库算法工作。同样的，它提供便捷的工厂函数 make_counting_iterator()。

示范 counting_iterator 基本用法的代码如下：

```
#include <boost/iterator/counting_iterator.hpp>
using namespace boost;
int main()
{
  counting_iterator<int> i(100);       //把 int 适配成计数迭代器

  assert(*i++ == 100);                 //后++操作
  assert(*i == 101);
  assert(*++i == 102);                 //前++操作

  vector<int> v;
  std::copy(                           //使用 copy 算法
      make_counting_iterator(0),       //从 0 填充到 10
      make_counting_iterator(10),
      back_inserter(v)                 //插入迭代器适配器
      );
}
```

下面的代码把随机数发生器包装成了一个可递增的类型 rand_int，它符合 counting_iterator 对 Incrementable 的要求：可递增、可拷贝构造、可赋值、可比较：

```
template<typename R>                           //要求类型是随机数发生器
class rand_int
{
private:
  R &r;                                        //随机数的引用
  int count;                                   //个数统计，用于相等比较
public:
  rand_int(R& _r, int c = 0):                  //构造函数
      r(_r), count(c) {}
  rand_int(rand_int const &other):             //拷贝构造函数
      r(other.r), count(other.count){ }
  void operator=(rand_int const &other)        //赋值函数
  {
      r = other.r;
      count = other.count
  }
  void operator++()                            //递增操作，增加计数
  {   ++count;}
  //比较操作，比较计数数量
  friend bool operator==(rand_int const &l, rand_int const &r)
  {   return l.count == r.count;}
  //转型到整数的操作符，返回随机数
  operator typename R::result_type () const
  {   return r(); }
};
```

现在，rand_int 就可以被用于 counting_iterator，适配后可以向任意的容器填充随机数：

```
int main()
{
  typedef counting_iterator<rand_int<rand48>,     //定义为单遍迭代器
      boost::single_pass_traversal_tag, int> RandIter;

  rand48 r;                                       //一个随机数发生器
  rand_int<rand48> r1(r, 0), r2(r, 10);           //包装为可递增类型
  RandIter first(r1), last(r2);                   //适配为计数迭代器
```

```
    vector<int> v;
    std::copy(first, last, back_inserter(v));      //使用 copy 算法
    assert(v.size() == 10);
}
```

counting_iterator 的另一个使用例子可见 3.6.8 小节。

3.6.6　函数输入迭代器

函数输入迭代器(function_input_iterator)很类似发生器迭代器(3.6.2 小节)，同样能够把一个函数或者函数对象适配成输入迭代器，但不同的是可以使用一个状态参数设定迭代器的起点和终点(有界)，因而更容易使用，它位于头文件<boost/iterator/function_input_iterator.hpp>。

类摘要

function_input_iterator 使用了模板元编程技术，类摘要如下：

```
template <class Function, class Input>
class function_input_iterator
    : public mpl::if_<                                    //注意模板元编程的使用
        function_types::is_function_pointer<Function>,
        impl::function_pointer_input_iterator<Function,Input>,
        typename mpl::if_<
            function_types::is_function_reference<Function>,
            impl::function_reference_input_iterator<Function,Input>,
            impl::function_input_iterator<Function,Input>
        >::type
    >::type
{
    typedef some_define base_type;
public:
    function_input_iterator(Function & f, Input i)    //构造函数
        : base_type(f, i) {}                          //初始化基类
};
```

function_input_iterator 使用 10.4 小节的 function_types 库来提取模板参数 Function 的类型信息，再使用 mpl 模板元编程技术进行分支决策（参见 11.3 小节），

在编译期决定从那个基类继承，而它本身并没有实际的功能。

function_input_iterator 在名字空间 boost::impl 里定义了三个具体实现类，分别是 function_pointer_input_iterator、function_reference_input_iterator 和 function_input_iterator，它们分别对应函数指针、函数引用和函数对象三种情形，这三个实现类的代码很类似，故下面仅以 boost::impl::function_input_iterator 为例。

boost::impl::function_input_iterator 的实现代码如下：

```
template <class Function, class Input>
class function_input_iterator                      //位于 boost::impl 名字空间
    : public iterator_facade<
    function_input_iterator<Function, Input>,
    typename Function::result_type,                //值类型
    single_pass_traversal_tag,                     //单遍迭代器
    typename Function::result_type const &         //常引用，可读不可写
    >
{
public:
    function_input_iterator() {}                   //构造函数
    function_input_iterator(Function & f_, Input state_ = Input())
        : f(&f_), state(state_), value((*f)()) {} //初始化成员变量

    void increment() {                             //递增操作
        value = (*f)();                            //调用函数产生值
        ++state;                                   //状态变化
    }

    typename Function::result_type const &         //解引用操作
    dereference() const
    {   return value;  }

    bool equal(function_input_iterator const & other) const  //相等比较
  {   return f == other.f && state == other.state; }         //比较状态值

private:
    Function * f;                                  //保存函数对象的指针
    Input state;                                   //状态
```

```
    typename Function::result_type value;          //解引用值
};
```

function_input_iterator 有两个模板参数：第一个 Function 与发生器迭代器一样，是一个具有 operator()的可调用物，被用来产生迭代器的解引用值；第二个 Input 要求是一个可递增、可比较、可缺省构造和拷贝构造的类型，也就是说支持 operator++和 operator==，它被用来执行单遍迭代器的相等比较操作。

用法

function_input_iterator 用起来很像发生器迭代器和计数迭代器的混合体，一方面它可以把函数或函数对象适配为一个迭代器，另一方面它又能够使用额外的 state_参数进行计数，在到达终点时自动停止。它的便捷的工厂函数是 make_function_input_iterator()，有两种重载形式，分别针对函数指针和函数引用。

示范 function_input_iterator 基本用法的代码如下：

```
#include <boost/iterator/function_input_iterator.hpp>
#include <boost/random.hpp>
using namespace boost;
int main()
{
  rand48 rng;                                    //rand48 随机数发生器

  std::copy(                                     //使用 std::copy 算法
      make_function_input_iterator(rng, 0),      //从 0 开始
      make_function_input_iterator(rng, 5),      //到 5 结束
      ostream_iterator<int>(cout, "\n")          //标准流输出 5 个随机数
      );
}
```

读者可以把这段代码与 3.6.2 小节发生器迭代器的示范代码对比一下，这里因为可以自行控制迭代器的起点和终点状态，所以可以使用 std::copy 算法。

还应该注意 function_input_iterator 构造函数中的状态参数的使用，这里用的是最简单最常用的 int 类型，它当然满足可递增、可比较等要求，但也可以使用任意可递增可比较的类型，例如迭代器：

```
  rand48 rng;                                        //rand48 随机数发生器
  vector<int> v(10);                                 //一个大小为 10 的标准容器
```

```
std::copy(                                          //使用 std::copy 算法
    make_function_input_iterator(rng, v.begin()),   //使用迭代器定义起点
    make_function_input_iterator(rng, v.end()),     //使用迭代器定义终点
    v.begin()   );                                  //拷贝到标准容器
```

上面的代码使用了 vector 的迭代器作为 function_input_iterator 的状态参数，用随机数填满了容器，而无须知道容器的确切大小。

function_input_iterator 还提供了一个特别的状态类型 infinite，它永远不会到达终点：

```
struct infinite {
    infinite & operator++() { return *this; }
    infinite & operator++(int) { return *this; }
    bool operator==(infinite &) const { return false; };          //注意
    bool operator==(infinite const &) const { return false; };    //注意
};
```

infinite 通过 operator==返回 false 导致状态比较总是不相等，因而函数输入迭代器可以无限地迭代产生值。

3.6.7 函数输出迭代器

function_output_iterator 可以把一个单参函数或函数对象适配成一个标准的**输出迭代器**，它位于头文件<boost/function_output_iterator.hpp>。

类摘要

function_output_iterator 是一个比较特殊的迭代器，并没有使用 iterator_adaptor 或 iterator_facade，类摘要如下：

```
template <class UnaryFunction>
class function_output_iterator
{
public:
  typedef std::output_iterator_tag iterator_category;    //输出迭代器分类
  typedef void                     value_type;
  typedef void                     difference_type;
  typedef void                     pointer;
  typedef void                     reference;
```

```
  explicit function_output_iterator();
  explicit function_output_iterator(const UnaryFunction& f);

  output_proxy operator*();                   //解引用操作
  function_output_iterator& operator++();
private:
  UnaryFunction m_f;                          //函数对象
};
```

function_output_iterator 的模板参数要求是可接受一个参数的单参可调用物，函数或者函数对象都可以，构造函数把函数保存在 private 成员变量 m_f 里，因此要求 UnaryFunction 必须是可拷贝构造和可赋值的。

解引用操作 operator*()是 function_output_iterator 的核心功能所在，它返回一个代理对象 output_proxy，把赋值操作转化为对函数的调用，因此*iter = t 就相当于 m_f(t)。

用法

使用 function_output_iterator 再配合标准算法 std::copy，我们就可以很容易地操作存储在容器中的所有元素，只需要把操作函数适配成迭代器即可。

下面的代码定义了一个转换 ASCII 码到十六进制数的函数对象 to_hex，它逐个地接受字符，再把它们转换成十六进制数存储在一个外部的 vector 中：

```
class to_hex
{
private:
  vector<unsigned char> &v;            //存储十六进制数的容器
  int count;                           //字符计数
  char trans(const char c) const       //ascii 转换十六进制数
  {
      if (c >= 'a')
      {   return c - 'a' + 10;}
      else if (c >= 'A')
      {   return c - 'A' + 10;}
      else
      {   return c - '0';}
  }
```

```
public:
  to_hex(vector<unsigned char> &_v): //构造函数
      v(_v), count(0){}
  void operator()(const char c)
  {
      static char tmp;
      if ((count++) % 2 == 0)
      {
          tmp = trans(c)* 0x10;
      }
      else
      {
          tmp += trans(c);
          v.push_back(tmp);
      }
  }
};
```

使用 function_output_iterator 适配 to_hex 后，我们就可以很容易地实现字符串到十六进制数（base16）的转换：

```
int main()
{
  char s[] = "1234abcd";                          //base16 编码的字符串

  vector<unsigned char> v;                        //存储十六进制数的容器
  to_hex h(v);                                    //创建函数对象
  function_output_iterator<to_hex> foi(h);        //适配成迭代器

  std::copy(s, s + 8, foi);                       //调用 copy 算法，输出到函数对象
  assert(v.size() == 4);
}
```

使用工厂函数 make_function_output_iterator()函数输出迭代器的创建也可以整合到 copy 算法中，用法更简单，代码如下：

```
  std::copy(s, s + 8,                                    //copy 算法
      make_function_output_iterator(to_hex(v))); //工厂函数创建迭代器
```

3.6.8 过滤迭代器

filter_iterator 可以选择性地迭代序列，“筛选”出所需要的元素，如何选择则需

要用一个谓词（返回 bool 值的函数或函数对象）来决定，相当于对原序列提供了一个“子视图”，它位于头文件<boost/iterator/filter_iterator.hpp>。

类摘要

filter_iterator 的类摘要如下：

```
template <class Predicate, class Iterator>
class filter_iterator:
  public iterator_adaptor<...>        //适配的细节省略
{
 public:
    filter_iterator();
    filter_iterator(Predicate f, Iterator x, Iterator end = Iterator());
    filter_iterator(Iterator x, Iterator end = Iterator());
    Predicate predicate() const;

    Iterator end() const;
    Iterator const& base() const;
    reference operator*() const;
    filter_iterator& operator++();
private:
    Predicate m_pred;              //谓词
    Iterator m_iter;               //迭代器
    Iterator m_end;                //迭代器终点
};
```

filter_iterator 需要两个模板参数，第一个参数 Predicate 是过滤条件谓词，用于过滤元素，只有满足条件（Predicate(x)==true）才会被选择；第二个参数 Iterator 是被适配的迭代器类型，应该满足可读和单遍迭代器概念，filter_iterator 将使用它来迭代。

filter_iterator 的构造函数一般需要传递谓词 Predicate（如果是可缺省构造的那么也可以不必传入）和迭代器，因为在选择元素时可能造成迭代器越界，因此除传入迭代起点外还必须传入迭代终点。

用法

作为示范，我们使用 filter_iterator 来迭代筛选某个整数区间的质数。

首先我们定义一个谓词函数 is_prime()：

```
bool is_prime(int x)          //判断整数x是否是质数
{
  for (int i = 2; i < x / 2; ++i)
  {
      if (x % i == 0)
      {   return false;}
  }
  return true;
}
```

假设我们要筛选 10-100 这个区间的整数，那么就可以考虑使用 counting_iterator 来计数生成这些整数。又因为所有的偶数都不是质数，所以又可以用 3.4.3 小节定义的步进迭代器 step_iterator 来跳过所有的偶数，形成一个嵌套形式的迭代器，最后再交给 filter_iterator 过滤。实现代码如下：

```
int main()
{
  typedef counting_iterator<int> ci_t;    //计数迭代器
  ci_t c1(11), c2(101);         //因为使用step_iterator迭代，故区间必须是奇数端点
  typedef step_iterator<ci_t> si_t;       //步进迭代器，在计数迭代器上步进
  si_t si1(c1), si2(c2);

  //定义过滤迭代器，注意函数指针类型的写法
  typedef filter_iterator<bool(*)(int), si_t> fi_t;
  fi_t first(&is_prime, si1, si2);        //迭代器起点，si2防止迭代越界
  fi_t last(&is_prime, si2, si2);         //迭代器终点，si2防止迭代越界

  std::copy(first, last,                  //调用copy算法
      ostream_iterator<int>(cout, " ")); //流迭代器输出
}
```

同样的，使用工厂函数 make_filter_iterator() 可以无须定义迭代器的类型，简化代码：

```
std::copy(
  make_filter_iterator(&is_prime, si1, si2),
  make_filter_iterator(&is_prime, si2, si2),
  ostream_iterator<int>(cout, " "));
```

3.6.9 转换迭代器

transform_iterator 把一个单参函数或函数对象应用于迭代的序列，解引用时使用函数来操作序列中的元素，效果与标准库的 transform 或 for_each 算法颇类似。它位于头文件<boost/iterator/transform_iterator.hpp>。

类摘要

transform_iterator 的类摘要如下：

```
template <class UnaryFunction,          //单参函数对象
          class Iterator,               //被适配的迭代器
          class Reference = use_default,
          class Value = use_default>
class transform_iterator:
  public iterator_adaptor<...>          //适配的细节省略
{
public:
  transform_iterator();
  transform_iterator(Iterator const& x, UnaryFunction f);

  UnaryFunction functor() const;
  Iterator const& base() const;
  reference operator*() const;
  transform_iterator& operator++();
  transform_iterator& operator--();
private:
  Iterator m_iterator;                  //迭代器
  UnaryFunction m_f;                    //函数对象
};
```

transform_iterator 需要两个基本的模板参数：UnaryFunction 是一个单参的可调用物，同 function_output_iterator 一样要求是可拷贝构造和可赋值的；第二个参数 Iterator 则要求满足可读迭代器概念。

transform_iterator 的核心操作是 operator*()，它变动了 operator*所需的 dereference()成员函数，解引用原迭代器再调用 UnaryFunction 操作，相当于 m_f(*m_iterator)，然后返回函数对象的转换结果（返回值）。

用法

transform_iterator 的用法和效果都非常像 std::transform 算法，可以在一个序列上迭代操作其中的所有元素，施加某些变动。

下面的代码先使用计数迭代器给容器赋初值，然后用 transform_iterator 把所有元素加上 5 输出到 cout，函数对象使用了 boost::bind 和 std::plus：

```
#include <boost/iterator/transform_iterator.hpp>
using namespace boost;
int main()
{
  typedef counting_iterator<int> ci_t;

  //使用计数迭代器向容器填充 10 个整数
  vector<int> v;
  std::copy(ci_t(0), ci_t(10), back_inserter(v));

  //使用 boost::bind 创建函数对象，把整数加 5
  std::copy(
      make_transform_iterator(v.begin(), bind(plus<int>(), _1, 5)),
      make_transform_iterator(v.end(), bind(plus<int>(), _1, 5)),
      ostream_iterator<int>(cout, " "));
}
```

这段代码对应的 transform 算法代码如下：

```
std::transform(
  v.begin(),v.end(),
  ostream_iterator<int>(cout, " "),
  bind(plus<int>(), _1, 5));
```

个人认为，transform_iterator 的 copy 算法用法更具有一致性，更容易理解。

3.6.10 索引迭代器

permutation_iterator 虽然直接翻译为“排序迭代器”，但它实际上并不执行真正的排序操作，只是改变了原有序列的索引顺序从而变动了迭代顺序。它位于头文件 <boost/iterator/permutation_iterator.hpp>。

类摘要

permutation_iterator 的类摘要如下：

```
template< class ElementIterator, class IndexIterator>
class permutation_iterator :
  public iterator_adaptor< permutation_iterator, IndexIterator...>
{
public:
  permutation_iterator();
  explicit permutation_iterator(ElementIterator x, IndexIterator y);

  reference operator*() const;
  permutation_iterator& operator++();
  ElementIterator const& base() const;
private:
  ElementIterator m_elt_iter;

  reference dereference() const       //注意这里
    { return *(m_elt_iter + *this->base()); }
};
```

permutation_iterator 的第一个模板参数 ElementIterator 并不用于迭代，它只是作为一个迭代器基准来检索数据；第二个模板参数 IndexIterator 是排序索引所在的迭代器，它才是被 iterator_adaptor 适配的迭代器，定义了 permutation_iterator 迭代的区间和顺序。

permutation_iterator 的核心操作是 dereference()，它先对 IndexIterator 解引用，获得索引值，然后把索引值加上 ElementIterator 再解引用。

因为使用了迭代器的算术运算，permutation_iterator 要求 ElementIterator 必须满足随机访问遍历迭代器概念，而 IndexIterator 解引用返回的值应该可转换为 ElementIterator 的距离类型。

用法

permutation_iterator 使用 IndexIterator 为 ElementIterator 定义了一个区间，并且使用索引值重新排列了元素的顺序。区间的大小不一定与原序列相同，可以是一个子区间，区间中元素也可以重复。

示范 permutation_iterator 用法的代码如下：

```
#include <boost/iterator/permutation_iterator.hpp>
using namespace boost;

int main()
{
  char s[] = "abcdefg";               //元素序列，有 7 个元素
  int idx[] = {6, 0, 2, 2, 4};        //索引序列，5 个索引，其中两个重复

  //typedef permutation_iterator<char*, int*> pi_t;
  std::copy(
      make_permutation_iterator(s, idx),
      make_permutation_iterator(s, idx + 5),
      ostream_iterator<char>(cout, " ")
      );
}
```

这段代码中定义了一个字符串序列 s 和一个索引序列 idx，idx 使用索引重新排列了 s 中的部分元素，然后我们使用工厂函数 make_permutation_iterator() 创建了两个索引迭代器。代码的运行结果如下：

```
g a c c e
```

它们分别是字符串 s 中的第 6、第 0、第 2、第 2 和第 4 个元素。

3.6.11 组合迭代器

zip_iterator 使用 tuple①对多个迭代器“打包”，可以同时移动所有被打包的迭代器，解引用 zip_iterator 将返回多个迭代器解引用结果的 tuple，它位于头文件 <boost/iterator/zip_iterator.hpp>。

类摘要

zip_iterator 使用了模板元编程技术，简化的类摘要如下：

```
template<typename IteratorTuple>
class zip_iterator :
```

① tuple 即元组，是一个可以容纳不同类型元素的数据结构，见推荐书目[1]。

```
  public iterator_facade<...>        //适配的细节省略
{
public:
  zip_iterator();
  zip_iterator(IteratorTuple iterator_tuple);
  const IteratorTuple& get_iterator_tuple() const;

  reference operator*() const;
  zip_iterator& operator++();
  zip_iterator& operator--();

private:
  IteratorTuple m_iterator_tuple;
};
```

zip_iterator 只有一个模板参数 IteratorTuple，它是一个迭代器引用类型的 tuple。tuple 里可以包含有任意多个迭代器，这些迭代器都应该满足可读迭代器的概念，zip_iterator 的 operator*()将返回 IteratorTuple 中的迭代器各自解引用后的 tuple。

由于 zip_iterator 组合了多个迭代器，构造它比较麻烦，通常需要使用工厂函数 make_zip_iterator()和 make_tuple()来简化创建过程。make_zip_iterator()使用一个迭代器的 tuple 来创建 zip_iterator，代码如下：

```
template<typename IteratorTuple>
zip_iterator<IteratorTuple>           //返回 zip_iterator
make_zip_iterator(IteratorTuple t)    //参数是一个 tuple
{ return zip_iterator<IteratorTuple>(t); }
```

用法

zip_iterator 最常见的应用场景是需要同时对多个序列进行迭代操作，然后把这些迭代结果组合起来。为了操作组合后的迭代结果，通常需要编写一个单参函数对象，它的参数也是 tuple，然后我们就可以使用 for_each、transform 算法或者 transform_iterator 来迭代 zip_iterator 得到最终的计算结果。

下面我们使用 zip_iterator 来实现 3.6.6 小节的 ASCII 码转换十六进制数的功能，读者可对比两者实现的异同。

首先我们需要定义一个新的函数对象 to_hex2，它的 operator()使用 tuple 作为参数：

```
class to_hex2
{
private:
  vector<unsigned char> &v;
  char trans(const char c) const
  { ...}                              //同 to_hex 的实现
public:
  to_hex2(vector<unsigned char> &_v):   //构造函数
    v(_v){}

  //定义 tuple 类型
  typedef boost::tuple<const char&, const char& > Tuple;
  void operator()(Tuple const& t) const //用 tuple 接受多个参数
  {
        static char tmp;                //代码与 to_hex 的类似
        tmp = trans(t.get<0>())* 0x10;
        tmp += trans(t.get<1>());
        v.push_back(tmp);
  }
};
```

接下来我们还需要使用 3.4.3 小节定义的步进迭代器 step_iterator 来在字符串上跳跃迭代奇偶位置，并在 for_each 算法中使用 zip_iterator，嵌套工厂函数来创建迭代器，代码如下：

```
#include <boost/iterator/zip_iterator.hpp>
using namespace boost;
int main()
{
  char s[] = "1234aBcD";                          //base16 编码字符串
  vector<unsigned char> v;                        //存储 16 进制数的容器

  typedef step_iterator<const char*> si_t;        //使用步进迭代器

  for_each(                                       //for_each 算法
      make_zip_iterator(                          //使用工厂函数简化代码
          make_tuple(si_t(s), si_t(s + 1))),      //两个起点
      make_zip_iterator(
          make_tuple(si_t(s + 8), si_t(s + 9))),  //两个终点
      to_hex2(v)                                  //传递以 tuple 为参数的函数对象
```

```
        );
    assert(v.size() == 4);
}
```

for_each 算法也可以用 copy 算法搭配 function_output_iterator 替换，效果相同：

```
std::copy(
    make_zip_iterator(                            //组合迭代器
        make_tuple(si_t(s), si_t(s + 1))),
    make_zip_iterator(                            //组合迭代器
        make_tuple(si_t(s + 8), si_t(s + 9))),
    make_function_output_iterator(to_hex2(v))     //函数输出迭代器
    );
```

3.7 总结

本章中我们深入研究了 Boost 对 C++迭代器概念的重要贡献。

我们首先回顾了迭代器设计模式，它是 C++和其他所有编程语言迭代器实现的理论基础。然后我们讨论了标准库和 Boost 定义的迭代器分类，不同的迭代器其能力上有很大的区别，了解这些区别有助于我们编写正确的代码。标准库定义了五类迭代器，但这个分类并不完美，存在一些小的缺陷，而 Boost 定义的新式迭代器则弥补了标准库的错误，给出了一个更合理的解决方案。

next_prior 和 iterator_traits 是两个很小的工具，它们改进了标准库的迭代器工具，用起来更加容易。next_prior 组件基于 std::advance()实现了可任意前进和后退的 next()和 prior()函数；iterator_traits 基于 std::iterator_traits，把非标准元函数转换为标准元函数。虽然它们都只做了很少的工作，但的确可以简化迭代器的使用。

Boost 对迭代器的工作主要集中在 iterators 库，它提供两个重要的类：使用外观模式的 iterator_facade 和使用适配器模式的 iterator_adaptor，可以帮助程序员更容易地构造符合标准的迭代器，从而能够以统一的算法+迭代器的解法操作各种数据，而且这种解法十分优雅。

iterators 库提供了繁多的迭代器工具，特别是迭代器适配器，可以把它们任意组合（类似装饰模式）得到许多神奇的效果。这些迭代器工具可大致分为如下几个类别，有助于我

们更好地掌握它们的用途：

- 输入型迭代器　：发生器迭代器、计数迭代器和函数输入迭代器；
- 输出型迭代器　：函数输出迭代器；
- 解引用型迭代器：共享容器迭代器和间接迭代器；
- 排序型迭代器　：逆向迭代器和索引迭代器；
- 其他迭代器　　：过滤迭代器、转换迭代器和组合迭代器。

还要说的是，serialization 库（参见 9.9.4 小节）也使用 iterators 库实现了几个很有用的迭代器，可以作为本章学习的进一步参考。另外流处理（第 8 章）的操作手法与迭代器也十分类似，读者也可以对比研究。

第 4 章

函数对象

本章讨论现代 C++编程中的另一个重要角色：函数对象。

函数对象（function object）又称仿函数（functor），是一个定义了 operator() 的对象，可以像普通函数一样被调用，因为可以具有类的所有能力，所以又可以被称为“智能函数”，要比普通函数更加强大。

Boost 库中包含很多功能强大的函数对象，其中一些已经在推荐书目[1]中做过介绍。本章中我们仅研究其中的四个函数对象：首先是 hash，它用于计算对象的散列值，被用来实现各种散列容器；然后是 mem_fn，是对标准库 std::mem_fun 和 std::mem_fun_ref 的增强；第三个是泛化的工厂函数对象 factory，它是一个“智能 new”，完全可以代替 new 关键字；最后我们会看到 forward，它是一个函数对象适配器，可以把一个只接受左值（引用）的函数对象适配成可以接受右值的新函数对象。

这些函数对象都非常实用，读者很快就会发现它们的价值。

4.1 hash

hash 是 C++ tr1 技术草案中规定的散列函数的一个具体实现，它完全符合 tr1 的定义，可以计算任意 C++对象的散列值，主要被用于实现各种无序散列容器，如 boost::unoredered_set 和 boost::ptr_unoredered_set。某种意义上说，hash 库提供的功能很像 Java 中 Object.hashCode()方法。

hash 位于名字空间 boost，为了使用 hash 组件，需要包含头文件 <boost/functional/hash.hpp>，即：

```
#include <boost/functional/hash.hpp>
using namespace boost;
```

4.1.1 类摘要

hash 是一个非常简单的函数对象，类摘要如下[①]：

```
template <class T> struct hash
    : std::unary_function<T, std::size_t>         //标准单参函数对象
{
    std::size_t operator()(T const& val) const    //计算 val 的散列值
};
```

hash 符合 std::unary_function 概念，是一个单参函数对象，成员函数 operator() 接受一个类型 T 的引用值作为输入，计算得到一个类型为 size_t 的散列值返回。

因为 hash 库符合 tr1 草案和相应的扩展，故它对 C++数据类型的支持是非常全面的，包括：

- char/wchar_t；
- short/int/long 等各种整数类型；
- bool 类型；
- float/double 等浮点数类型；
- long long 和 long double（如果编译器支持）；
- 指针和数组；
- std::string；
- 以及标准库中的 pair、complex 结构和 vector、list 等标准容器。

hash 库不支持上述以外的其他类型，包括标准容器适配器（stack、queue、priority_queue）和所有 Boost 容器（array、bimap、circular_buffer 等）。

① 实际上 hash 函数对象使用模板特化有多种不同的形式，这里的类摘要只是其中较常用的一个，不同形式的区别主要在于 operator() 的参数类型。

4.1.2 用法

hash 是一个很简单的函数对象，因此很容易使用，只需要创建一个实例，然后像使用函数一样调用它的 operator()就可以了，例如[①]：

```
#include <boost/functional/hash.hpp>
using namespace boost;
int main()
{
  cout << hex;                                          //设定标准流使用十六进制格式

  cout << hash<int>()(0x2000) << endl;                 //计算整数的散列值
  cout << hash<double>()(1.732) << endl;               //计算浮点数的散列值
  cout << hash<const char*>()("string") << endl; //计算字符数组的散列值

  complex<double> c(1.0, 2.0);
  cout << hash<complex<double> >()(c) << endl;   //计算复数的散列值

  cout << hash<string>()("string") << endl;      //计算标准字符串的散列值

  vector<int> v(12);
  cout << hash<vector<int> >()(v) << endl;        //计算标准向量容器的散列值
  map<int, string> m;
  cout << hash<map<int, string> >()(m);            //计算标准映射容器的散列值
}
```

这段代码计算了各种基本类型和标准库容器的散列值，会输出一系列的十六进制数值。如果企图用 hash 计算不支持的类型，那么会导致编译错误（扩展 hash 使之支持其他类型参见 4.1.4 小节），例如：

```
array<int, 5> ar;                                    //Boost 提供的数组容器
cout << hash<array<int, 5> >()(ar) << endl;         //编译错误
```

除了直接计算对象的散列值，hash 更常见的用途是被用作无序容器的模板参数，作为容器计算散列值的一个策略对象：

① 注意：如果读者使用 STLport 作为 C++标准库的实现，那么需要当心 STLport 在 std 名字空间也声明了一个 hash 类，会出现名字冲突，如果必要请在 hash 前加上 boost 名字空间限定，即 boost::hash。

```
#include <boost/functional/hash.hpp>
#include <hash_set>                        //STLport 的无序散列容器
#include <hash_map>
#include <boost/unordered_set.hpp>         //boost 的无序散列容器
#include <boost/unordered_map.hpp>

int main()
{
  std::hash_set<double, boost::hash<double> > hs;
  std::hash_map<int, string, boost::hash<int> > hm;

  boost::unordered_set<int, boost::hash<int> > us;
  boost::unordered_map<int, string, boost::hash<int> > um;
}
```

上面的代码分别使用了 STLport 和 Boost 中实现的散列容器，并统一使用 hash 作为容器计算散列值的模板参数。

4.1.3　实现原理

hash 函数对象实际上只是一个简单的包装类，真正的散列值计算实现是在 boost 名字空间里的 hash_value()函数，而 hash_value()则又针对各种数据类型定义了不同的重载形式，最后 hash 再针对不同的类型使用模板特化来调用 hash_value()函数。

hash_value()函数的声明通常是下面的形式：

```
template <class T>
std::size_t hash_value(T const&);
```

例如，对于基本数据类型 int，hash 的实现如下：

```
inline std::size_t hash_value(int v)      //注意，这个不是模板函数，而是重载
{
    return static_cast<std::size_t>(v);   //计算散列值
}
template <> struct hash<int>              //模板特化
     : public std::unary_function<int, std::size_t>
{
    std::size_t operator()(int v) const
    {
```

```
        return boost::hash_value(v);        //调用散列函数
    }
};
```

因此，只要我们在自己的名字空间或者其他可被 ADL（参数依赖查找规则）查找到的位置中定义了 hash_value()的重载形式，就能够对自定义类型调用 hash 函数对象计算散列值。

4.1.4 扩展 hash

使用 hash 库提供的对基本数据类型计算散列值的能力和一些辅助函数，我们可以实现对自定义类型计算散列值。

简单的计算散列值

下面的代码定义了一个 person 类，在成员函数 hash_value()中计算散列值（也可以是其他的名字），然后再在外部（或者定义为内部友元函数）实现重载的自由函数 hash_value()：

```
class person
{
private:
  int id;
  string name;
  unsigned int age;
public:
  person(int a, const char* b, unsigned int c):      //构造函数
    id(a),name(b), age(c){}
  size_t hash_value() const              //自定义的散列计算函数
  {
      return hash<int>()(id);            //只调用 hash 函数对象根据 id 计算散列值
  }
};
size_t hash_value(person const & p)   //同名字空间重载 hash_value
{ return p.hash_value();}
```

这样就可以把 hash 函数对象应用于我们自定义的 person 对象了：

```
person p(1, "adam", 20);
```

```
cout << hash<person>()(p) << endl;        //可正确执行
```

如果要把 person 对象放入无序容器，除了实现 hash 外，我们还需要定义 operator==：

```
class person
{
  ... //同前
  friend bool operator==(person const & l, person const & r)
  {   return l.id == r.id;}
};

int main()
{
  unordered_set<person> us;                   //无序集合容器

  us.insert(person(1, "adam", 20));
  us.insert(person(2, "eva", 20));
  assert(us.size() == 2);
}
```

组合散列值

如果要对多个目标计算散列值，那么可以使用 hash 库提供的辅助函数 hash_combine()和 hash_range()来组合散列值，它们的声明如下：

```
template<typename T>
void hash_combine(size_t & seed, T const& v);
template<typename It>
std::size_t hash_range(It first, It last);
template<typename It>
void hash_range(std::size_t& seed, It first, It last);
```

hash_combine()使用一个变量 seed 作为初始输入参数，可以对多个变量连续调用，最终计算得到的散列值再从 seed 输出。散列值的计算与 hash_combine()的运算顺序有关，即使是对同样的元素，如果计算顺序不同，那么最后得到的散列值也会不同。

使用 hash_combine()可以为 person 类定制新的散列计算方法，代码如下：

```
size_t hash_value() const
{
  size_t seed = 2011;                     //可以是任意的整数
```

```
    hash_combine(seed, id);                //组合三个变量
    hash_combine(seed, name);
    hash_combine(seed, age);
    return seed;
}
```

hash_range()是另外一种组合散列值的方式，它对一个迭代器区间内的所有元素调用 hash_combine()计算散列值，如果不给初始 seed 值（两参数的形式），那么 seed 默认为 0。hash_range()的用法示例如下：

```
using namespace boost::assign;
vector<int> v = (list_of(1),2,5,8,15);
size_t hv = hash_range(v.begin(), v.end());

unordered_set<int> us = (list_of(1),2,5,8,15);
hv = hash_range(us.begin(), us.end());
```

灵活使用 hash_combine()和 hash_range()，我们就可以对任意 C++对象使用任意策略计算散列值（实际上 hash 库对标准容器的处理就用了这两个函数）。

下面的代码定义了一个类 demo_class，它有一个 int 成员 x 和一个 vector<string>成员 v，散列值的计算方式是用 0 作为 seed，先计算 x，然后再逆序计算 v 里的所有元素：

```
class demo_class
{
private:
    vector<string> v;
    int x;
public:
    size_t hash_value()
    {
        size_t seed = 0;                                  //seed 取值为 0
        hash_combine(seed, x);                            //先计算整数的散列值
        hash_range(seed, v.rbegin(), v.rend());           //逆序遍历 vector

        return seed;
    }
};
```

4.2 mem_fn

mem_fn 是对 C++98 标准库中定义的成员函数适配器的增强，同 boost::bind 一样，mem_fn 大大超越了 std::mem_fun 和 std::mem_fun_ref，可以调用对象或对象指针的任意成员函数或成员变量（标准库只能调用 const 成员函数），而且支持多达 8 个参数。

mem_fn 位于名字空间 boost，为了使用 mem_fn 组件，需要包含头文件 <boost/mem_fn.hpp>，即：

```
#include <boost/mem_fn.hpp>
using namespace boost;
```

4.2.1 工作原理

同 boost::bind 一样，mem_fn 也不是一个单一的模板函数，而是为了支持各种情况有许多的重载形式，它基本的声明形式如下：

```
template<class R, class T> mf<R, T> mem_fn(R T::*f);        //mem_fn 函数
template<class R, class T> class mf                         //辅助函数对象
{
public:
    R & operator()(T * p,...) const;                        //调用对象的指针
    R & operator()(T & t,...) const;                        //调用对象的引用
}
```

mem_fn 函数接受一个类型 T 返回类型为 R 的成员函数指针 f，然后返回一个持有成员函数指针的函数对象 mf。mf 的 operator()可以传入类型 T 的指针/引用以及若干参数，再转而调用其成员函数。

mem_fn 支持最多 8 个参数，这与 bind 的默认参数数量限制是一致的（bind 默认最多支持 9 个参数，但用于绑定成员函数时需要“牺牲”一个参数传递对象）。与 bind 不同的是mem_fn没有_1、_2这样的参数占位符，因此不能实现对参数的绑定，必须在operator()中传递所有的参数。

4.2.2 用法

如果读者熟悉 boost::bind，那么 mem_fn 会很容易学习，它就像是一个简化版的

bind，只能绑定类的成员函数或成员变量。

mem_fn 不仅支持普通对象，也支持对象指针和智能指针。对于智能指针，mem_fn 会使用自由函数 boost::get_pointer()[①]获取真正的指针，不会导致 std::auto_ptr 的所有权转移或者 boost::shared_ptr 的引用计数增加，因此非常安全。

假设我们有下面的一个类：

```
class demo_class
{
public:
  int x;                              //一个公开的成员变量
  demo_class(int a = 0):x(a){}        //构造函数
  void print()                        //一个无参的成员函数，非 const
  {
      cout << x << endl;
  }
  void hello(const char* str)         //单参成员函数，非 const
  {
      cout << str << endl;
  }
};
```

那么 mem_fn 可以这样使用：

```
#include <boost/mem_fn.hpp>
#include <boost/smart_ptr.hpp>
int main()
{
  demo_class d;
  mem_fn(&demo_class::print)(d);           //绑定普通对象，调用无参成员函数

  demo_class *p = &d;
  mem_fn(&demo_class::hello)(p, "hello");  //绑定对象指针，调用单参成员函数

  auto_ptr<demo_class> ap(new demo_class(100));
  mem_fn(&demo_class::print)(ap);          //绑定自动指针
  assert(ap.get() != 0);                   //自动指针的所有权没有转移
```

① get_pointer()是一个很小的辅助函数，它对多个智能指针类型做了重载，总可以获得真正的原始指针。

```
    shared_ptr<demo_class> sp(new demo_class);          //绑定共享指针
    mem_fn(&demo_class::hello)(sp, "world");
}
```

mem_fn 更常见的用法是作为标准算法的一个参数，对容器内的所有元素调用无参成员函数，无论容器中存储的是元素本身、指针或是智能指针都可以正常工作：

```
vector<demo_class> v(10);                          //标准向量容器
std::for_each(v.begin(), v.end(),                  //使用 for_each 算法
    mem_fn(&demo_class::print));                   //mem_fn 调用成员函数
```

这段代码使用了标准库的 for_each 算法，对 vector 中的每个元素调用了 print() 函数，用法就像标准库的 mem_fun 和 mem_fun_ref 一样，但完全不用考虑成员函数的 const 属性和元素是否为指针，mem_fn 自动处理了所有的问题。

mem_fn 不仅可以调用成员函数，它也能够选择类的（公开）成员变量，功能类似非标准函数对象适配器 select1st 和 select2nd，用法与调用成员函数相同，例如：

```
demo_class   d(1);
cout << mem_fn(&demo_class::x)(d) << endl;         //访问成员变量
```

4.2.3 其他议题

本小节讨论关于 mem_fn 的一些其他议题。

与标准库函数对象适配器的比较

C++98 标准库为成员函数提供了 mem_fun 和 mem_fun_ref 两个函数对象适配器，但使用上有很多限制：它们只能调用类的 const 成员函数，而且只能是无参调用。另外，这两个函数的命名也存在缺陷，mem_fun 作用于对象指针，而 mem_fun_ref 作用于普通对象，用起来经常会混淆。

mem_fn 统一了成员函数适配的用法，完全解决了这些问题。

配合其他 Boost 组件

mem_fn 可以配合 boost.ref 库、boost.typeof 库和 boost.function 库使用以发挥更大的作用：ref 库可以让 mem_fn 持有一个参数的引用而不是拷贝，typeof 和

function 则可以存储 mem_fn 生成的函数对象，供延后使用。

与 bind 的比较

很多情况下 bind 都可以代替 mem_fn，只需要增加一个_1 占位符用于传递对象即可。例如，之前的部分 mem_fn 的 bind 等价表达式是：

```
demo_class d;
bind(&demo_class::print, _1)(d);
bind(&demo_class::hello, _1, "hello")(&d);
```

如果类的成员函数有多个参数，通常 bind 的用法会更加灵活，因为它可以使用占位符绑定参数：

```
vector<demo_class> v(10);
std::for_each(v.begin(), v.end(),
  bind(&demo_class::hello, _1, "world"));           //调用有参成员函数
```

但 mem_fn 也有它自己的优势：更接近标准库的函数对象适配器，写法简单，容易理解和掌握。

调用约定支持

mem_fn 支持__stdcall、__cdecl 和__fastcall 的调用约定，只需要在包含头文件<boost/mem_fn.hpp>前定义宏 BOOST_MEM_FN_ENABLE_STDCALL、BOOST_MEM_FN_ENABLE_FASTCALL 或 BOOST_MEM_FN_ENABLE_CDECL，此外的用法没有任何区别。这些宏的具体用法请参考 Boost 文档。

4.3 factory

factory 是一个很小却很强大的库，它实现了工厂设计模式，用函数对象封装了对象的创建过程，消除了关键字 new 的随意使用，有助于改善程序的设计结构。

factory 位于名字空间 boost，为了使用 factory 组件，需要包含头文件<boost/functional/factory.hpp>，即：

```
#include <boost/functional/factory.hpp>
using namespace boost;
```

4.3.1　类摘要

factory 库提供了两个函数对象，factory 和 value_factory，我们将重点讨论 factory，value_factory 将在稍后介绍。

factory 的类摘要如下：

```
template< typename Pointer,
        class Allocator, factory_alloc_propagation >
class factory
{
 public:
    typedef typename boost::remove_cv<Pointer>::type   result_type;
    typedef typename boost::pointee<result_type>::type value_type;
     inline result_type operator()() const;
     ...//其他 operator()的定义
};
```

factory 是一个模板类，有三个模板参数，但通常我们只需要使用第一个模板参数 Pointer，后两个可以用缺省值。

模板参数 Pointer 是被创建出的“指针”类型。之所以模板参数是 Pointer 而不是 T*的原因是 factory 不仅支持创建原始指针，也能够创建智能指针，包括 std::auto_ptr 和 boost::shared_ptr。

factory 内部使用元函数定义了两个 typedef，可以被用于泛型编程：result_type 是函数对象调用后返回的类型，即创建的指针类型，value_type 则是指针指向的值类型。

operator()是 factory 的核心功能所在，它支持最多 10 个参数，这些参数被传递给 value_type 的构造函数用来创建对象。

4.3.2　用法

factory 封装了 new 操作符，基本相当于 new T()[1]，可以直接创建出 T*指针。如果说 checked_delete（2.2 小节）是“智能 delete，”那么 factory 就是一个“智能 new”。

使用 factory 要求被创建的类型 T 至少要有一个 public 构造函数，否则 factory

① 说“基本”是因为 factory 比 new 操作符功能更强大，还能够支持自定义内存分配方式。

会因为无法访问 protected 或者 private 的构造函数而不能完成创建工作。

无参创建指针

factory 可以完全代替 new，只需要在模板参数中指明要创建的指针类型，它就会如 new 一样完成工作。采用无参的形式时 factory 将调用 value_type 的缺省构造函数完成对象的初始化，例如：

```
int *pi = factory<int*>()();                    //注意，两对括号
string *ps = factory<string*>()();
pair<int, double> *pp = factory<pair<int, double>* >()();
```

请读者注意 factory 的调用方式。因为它不是函数而是函数对象，因此我们必须先用一对括号调用它的构造函数创建出 factory 的一个临时对象，然后再用第二对括号调用它的 operator()来创建所需的对象。

这三行代码对应的 new 表达式是：

```
int *pi = new int;
string *ps = new string;
pair<int, double> *pp = new pair<int, double>;
```

factory 创建出的指针可以用 delete 操作符删除，当然最好使用与它对应的“智能 delete”——checked_delete：

```
checked_delete(pi);
checked_delete(ps);
checked_delete(pp);
```

创建智能指针

factory 也可以创建智能指针对象，这样我们就无须关心指针的删除问题，智能指针会自动处理。创建智能指针时必须在 factory 的模板参数中写完全智能指针的类型，不必再加*号，因为它们本身就已经是“指针”。

下面的代码创建了 auto_ptr 和 shared_ptr 智能指针：

```
auto_ptr<int> ap = factory<auto_ptr<int> >()();
shared_ptr<string> sp = factory<shared_ptr<string> >()();
```

factory 不能创建 scoped_ptr，这是因为 scoped_ptr 不支持拷贝转移语义，下面

企图用 factory 创建 scoped_ptr 的代码将导致编译错误：

```
scoped_ptr<int> p = factory<scoped_ptr<int> >()();         //编译错误
```

带参数创建指针

factory 也支持使用多个（最多 10 个）参数来创建指针对象，这些参数将传递给类的构造函数完成初始化。但由于“技术上的原因”，带参数的创建功能存在一个小缺陷：要求参数必须是左值类型，如果传递右值会无法编译。请见下面的示范：

```
int a = 10, b = 20;                                //声明两个变量用来创建指针
int *pi = factory<int*>()(a);
string *ps = factory<string*>()("char* lvalue"); //字符串也是左值
pair<int, int>* pp = factory<pair<int, int>*>()(a, b);

//下面的代码使用了右值，无法通过编译
int *pi2 = factory<int*>()(10);
pair<int, int>* pp = factory<pair<int, int>*>()(1, 2);
```

这个技术缺陷大大限制了 factory 的使用——为了创建一个对象必须声明若干个临时变量实在是太麻烦了，而且很不优雅。不过好在这个缺陷并非不可弥补，使用 bind 库包装 factory 函数对象可以解决这个问题，因为 bind 对参数类型没有限制，它内部持有参数的拷贝，可以被用作左值。

factory 的 bind 用法要显得麻烦一些，语法上也比较复杂：

```
int *p = bind(factory<int*>(),10)();               //可以使用右值
```

这行代码首先创建了一个 factory<int*>的临时对象，使用 bind 为它绑定了一个值为 10 的参数。bind 函数生成了一个新的函数对象，因此需要再用一对圆括号来调用它的 operator()，然后 bind 把参数 10 转发给 factory 对象最终完成 int 指针的创建工作。

bind 表达式也可以写成如下的形式：

```
int *p = bind(factory<int*>(), _1)(10);            //使用占位符
```

这个 bind 表达式使用了占位符_1 来代替传入的参数，真正的参数在 operator()中被传入（有关 bind 的更多信息可参考推荐书目[1]）。

稍后的 4.4 小节的 forward 库可以用另一种方式来适配 factory 函数对象，使它能够使用右值。

4.3.3 value_factory

value_factory 是 factory 库提供的另外一个函数对象，它同样可以创建对象，用法与 factory 也很类似，但不同的是它创建出的不是指针，而是真正的对象实例，它位于头文件<boost/functional/value_factory.hpp>。

value_factory 的类摘要如下：

```
template< typename T >
class value_factory
{
public:
  typedef T result_type;
  inline result_type operator()() const;
  ...//其他 operator()的定义
};
```

value_factory 的声明与实现都要比 factory 简单，它仅有一个 typedef (result_type)，同样支持最多传入 10 个参数，也同样要求参数必须是左值，operator()将返回创建对象的拷贝。

下面的代码示范了 value_factory 的用法：

```
int i = value_factory<int>()();
string str = value_factory<string>()("hello");
BOOST_AUTO(p,(value_factory<pair<int, string>>()(i, str)));
```

左值的问题同样可以使用 bind 来解决，例如：

```
int i = bind(value_factory<int>(), 10)();
complex<int> c = bind(value_factory<complex<int> >(), _1, _2)(10, 20);
```

4.3.4 使用 typeof 库

factory 库可以利用 boost.typeof 库的推导赋值表达式的能力来简化代码的编写，这样可以不用把创建出的对象类型信息写两遍，例如：

```
//类型自动推导为 int*
BOOST_AUTO(pi, factory<int*>());
assert(typeid(pi) == typeid(int*));
```

```
//类型自动推导为 complex<int>*
BOOST_AUTO(pc, factory<complex<int>* >());
assert(typeid(pc) == typeid(complex<int>*));
```

如果 factory 要创建的类型有多个参数，那么我们必须要用一对圆括号把 factory 创建表达式括起来，或者使用 typedef，这是由于宏预处理本身的缺陷所决定的（使用逗号来分隔宏参数）。例如：

```
BOOST_AUTO(pp , (factory<pair<int, int>*>()(a, b)));
```

4.4 forward

forward库提供一个函数对象适配器 forward_adapter[①]，能够把一个只接受左值(引用）的函数对象适配成可以接受右值的新函数对象。需要注意的是它只能适配函数对象，不能适配函数指针（与 bind 不同)。

forward 位于名字空间 boost，为了使用 forward 组件需要包含头文件 <boost/functional/forward_adapter.hpp>，即：

```
#include <boost/functional/forward_adapter.hpp>
using namespace boost;
```

4.4.1 类摘要

forward_adapter 的类摘要如下：

```
template< class Function,
    int Arity_Or_MinArity, int MaxArity >
class forward_adapter
{
public:
    forward_adapter(Function const& f = Function());

    typedef Function        target_function_t;
```

① forward 库另外还有一个功能与用法相同的“轻量级”适配器 lightweight_forward_adapter，本书未做介绍。

```
    Function & target_function();
    unspecified operator(...);
};
```

forward_adapter 使用类适配器设计模式从第一个模板参数 Function 继承[①]，它实际上是持有了 Function 再转换成另一个接口，但我们可以使用 target_function()来获得被包装的函数对象。因为使用了 result_of 来推导 operator()的返回类型，因此 Function 必须满足 result_of 的要求，即必须符合函数对象的标准定义——内部有类型定义 result_type。

第二个模板参数 Arity_Or_MinArity 是一个整数非类型模板参数，它确定了 Function 的参数数量。如果再指定第三个模板参数，那么 Arity_Or_MinArity 和 MaxArity 共同确定了 Function 的参数数量范围：最少 Arity_Or_MinArity 个，最多 MaxArity 个。

forward_adapter 的后两个模板参数不是必须提供的，如果不明确写出 forward_adapter 也能够使用模板元编程正确实现所需的功能，但如果明确地写出参数数量将有助于编译器进行模板推演，减少元程序的计算时间。

forward_adapter 的实现动机和原理涉及到 C++中函数传递参数的语言细节：为了能够同时使用左值和右值，函数的入口参数必须同时提供 T&和 T const &两种形式。forward_adapter 基于这个原理使用预处理元编程实现了从右值到左值的转换，提供了针对所有参数不同形式的函数重载。为了避免预处理过于复杂，forward_adapter 缺省支持最多六个参数，但可以使用宏 BOOST_FUNCTIONAL_FORWARD_ADAPTER_MAX_ARITY 改变。

4.4.2 用法

forward_adapter 可以把一个只接受左值的函数对象适配成既可以接受左值也可以接受右值的新函数对象，就像是对原函数对象加了一层薄薄的包装。不过包装后的新函数对象不完全符合标准，没有内部的 result_type 定义（这是否是一个失误？），如果要获得函数对象的返回值类型需要通过 target_function_t 来间接获得，即：

```
forward_adapter<F>::target_function_t::result_type
```

如果不是非常关心返回值类型，也可以直接使用 BOOST_AUTO 宏。

① 正是这个原因导致 forward_adapter 不能适配函数指针，因为指针类型无法被用于继承。

下面的代码示范了 forward_adapter 应用于标准函数对象：

```
#include <boost/functional/forward_adapter.hpp>
int main()
{
  //适配标准库的比较大小函数对象
  typedef forward_adapter<greater<int>> fa_t;
  assert((is_base_of<greater<int>, fa_t >::value));

  //获得返回值类型
  typedef fa_t::target_function_t::result_type result_type;
  assert((is_same<result_type, bool>::value));

  result_type b = fa_t()(1, 2);      //因为 greater<int>支持右值，所以也可写成
                                     //fa_t().target_function()(1, 2)
  assert(!b);
}
```

对于不可拷贝的函数对象（例如从 boost:: noncopyable 继承），forward 的模板参数可以使用引用类型：

```
struct functor: boost::noncopyable          //一个不可拷贝的函数对象
{
  typedef void result_type;

  template<typename T>
  void operator()(T& x) const
  {    cout << x << endl;}
};
int main()
{
  typedef forward_adapter<functor& > fa_t;   //使用引用类型

  functor f;                                 //函数对象实例
  fa_t fa(f);                                //适配函数对象
  fa(2);                                     //调用成功
  fa.target_function()(2);                   //编译错误，不支持右值
}
```

使用 forward_adapter 也可以很容易地适配 factory 函数对象，为它增加对左值的

支持，更加方便使用[①]。

```
template<typename T>
class factory_ex:                              //定义 factory 的 forward 适配器类
  public forward_adapter<factory<T> >     //可视为元函数转发
{};

int main()
{
  int *x = factory_ex<int*>()(10);         //适配后可以直接使用右值
  pair<int, int>* pp = factory_ex<pair<int, int>*>()(1, 2);
}
```

4.5 总结

本章内容较少，只介绍了四个函数对象。因为函数对象只能用于运行时，所以没有涉及太多的模板元编程知识，相信学习起来会轻松一些。

hash 是一个用来计算散列值的函数对象，符合 C++ tr1 规范，可以被用于各种散列容器。它的实现原理很简单，使用多个重载或模板特化的 hash_value()函数来计算不同类型的散列值，并且提供了辅助函数 hash_combine()和 hash_range()来组合散列值，所以我们可以定制自有类型的散列值计算策略，然后把它容纳在散列容器中。

mem_fn 是对标准库成员函数适配器 std::mem_fun 和 std::mem_fun_ref 的增强，可以适配任意的成员函数，并没有标准成员函数适配器那么多的限制，用法更简单。因为它非常有用，所以被收入了 C++ tr1 草案，出现在 C++11 中。不过由于有了更好的函数绑定器 boost::bind，所以更多的时候使用 bind 会更灵活方便，mem_fn 只能适用于比较简单的场合。

本章最后我们研究了 factory 和 forward，这两个函数对象经常会连起来使用，可以

① 也可以使用工厂函数的方式，例如：

```
template<typename T> inline
forward_adapter<factory<T> >
get_factory()
{
    typedef forward_adapter<factory<T> > factory_t;
    return factory_t();
}
```

代替操作符 new 创建对象，是一个“智能 new”。也许有的读者会认为 factory 的用法很累赘麻烦，但这实际上是没有领会设计模式的精髓——封装，factory 可以更好地隔离硬式创建对象的代码，具有更好的类型安全性。而且某种意义上 new 是和 delete 一样的“麻烦存在”，既然强调使用 checked_delete 来消灭 delete 的使用，那也应该对称地使用 factory 来消灭 new 的使用。

第 5 章

指针容器

容器是 STL 中最引人注意的部分，泛型的容器可以容纳任意的元素，免去了手工构造数据结构的麻烦，vector、list、set 等标准容器已经成为了广大程序员必不可少的工具。

Boost 延续了标准库的思想，进一步扩充了容器的范围，为 C++社区提供了更多的容器。推荐书目[1]中已经讨论了 Boost 的大部分新式容器，如 array、dynamic_bitset、any、multi_array 等。从本章开始我们将看到另外三类容器：指针容器、侵入式容器和多索引容器。这三个容器基本概念上与标准容器非常相似，但它们都在不同的方向上作出了扩展，丰富了我们的容器工具箱。

本章我们先研究指针容器库 ptr_container，它可以安全地容纳指针作为容器元素，并保证没有内存泄漏的风险。

5.1 概述

很多时候我们需要在容器中存储指针而不是元素本身（比如元素不满足标准容器的要求，存储抽象类而不是具体类，避免值语义内存拷贝的代价），但直接存储原始指针手法太初级，很不安全也难于管理，我们可以有如下的替代选择：

（1）使用 shared_ptr，它可以在容器中很好地管理指针，是最通用的解决方案，但由于容器的拷贝语义，使用存储的 shared_ptr 时需要进行引用计数的增减，操作效率会略微降低，而且使用迭代器操作容器内的 shared_ptr 也显得不太方便（使用 3.6.4 小节的间接迭代器可获得一定程度上的改善）。

（2）使用内存池 boost.pool。这种方法类似于自行创建一个小型的垃圾回收机制，可

以任意分配内存而不必担心回收，创建的对象可以直接放到标准容器中，不必在容器销毁时使用 delete 删除指针，但这种方法也有缺点，因为目前在 C++中暂时还没有一个“泛用”的内存池（垃圾回收机制），boost.pool 必须为每一类对象创建一个单独的内存池，使用起来很麻烦[①]。

（3）第三种容纳指针的方法就是使用 Boost 提供的指针容器库 ptr_container（下文中如不做特别声明，“指针容器”即是指 boost.ptr_container），它实现了数个标准容器风格的可容纳指针的容器，而且是高效和异常安全的，在某些必须保存指针的情况下特别有用。

ptr_container 库位于名字空间 boost，由数个不同的头文件组成，它们都位于目录 <boost/ptr_container/>下，使用具体的容器时包含所需头文件即可。

5.1.1　入门示例

ptr_container 库在开发时特意模仿了标准容器的风格，很多地方都很像标准容器，因而比较容易学习，但毕竟它容纳的不是普通元素而是指针，所以我们需要留意其特别之处。

本小节中我们先通过两个简单的小例子来快速预览一下指针容器的功能、用法和特性。

示例 1

第一个例子使用了类似 std::vector 的向量指针容器 ptr_vector，它演示了指针容器与标准容器的相同之处：

```
#include <boost/ptr_container/ptr_vector.hpp>    //向量指针容器头文件
using namespace boost;

int main()
{
  typedef ptr_vector<string> ptr_vec;              //一个容纳 string 的指针容器
  ptr_vec vec;                                     //声明一个指针容器实例

  vec.push_back(new string("123"));                //添加两个元素
  vec.push_back(new string("abc"));                //注意我们添加的是 new 创建的指针
```

① 使用内存池的另外一个副作用是程序员失去了对内存的控制，不能利用 RAII 的好处，内存释放的时机只能由内存池来控制，或者使用内存池的方法来强制释放内存调用析构函数，但这种操作实质上与使用 delete 没有本质差别。

```
    assert(!vec.empty());           //可以使用类似标准容器的 empty()函数
    assert(vec.size() == 2);        //同样可以使用 size()获得容器的大小

    assert(vec[0] == "123");        //使用 operator[]访问元素，注意返回的是引用
    vec.back() = "def";             //使用 back()成员函数访问末尾元素,同样返回的是引用
    assert(vec[1] == "def");

    ptr_vec::iterator iter = vec.begin();       //获取容器的迭代器
    assert(iter->length() == 3);    //迭代器可以直接操作指针指向的内容

    vec.clear();                    //可以使用 clear()清空容器，内存被安全释放
    assert(vec.empty());
}
```

这段代码看起来与标准容器 std::vector 的用法非常接近，但有一点本质的不同：我们向容器添加的元素是用操作符 new 动态创建的对象，容器容纳的是指针而不是元素本身（或者拷贝）。虽然指针容器中存储的是指针，但我们在使用时却感觉不到指针的存在，好像它里面容纳的就是普通的元素——这是因为指针容器在内部替我们多做了一次解引用操作（operator*），所以用起来非常方便，减少了操作原始指针可能出现的错误。

下面对比一下使用标准容器存储指针的用法：

```
vector<string*> vec;                    //使用 vector 存储指针,注意模板参数有*号
vec.push_back(new string("123"));       //添加指针
vec.push_back(new string("123"));

assert(*vec[0] == "123");               //获得元素后需要使用 operator*解引用
*vec.back() = "def";
assert(*vec[1] == "def");

vector<string*>::iterator iter = vec.begin();
assert((*iter)->length() == 3);         //迭代器同样需要使用 operator*
```

很明显，使用标准容器存储指针时访问元素要麻烦许多，同时我们还必须自行管理对象的生存周期，使用完毕后要记得删除指针，否则就会造成内存泄漏。

示例 2

接下来我们研究第二个例子，它演示了指针容器的指针所有权转移，这是与标准容器显著不同的地方：

```
#include <boost/ptr_container/ptr_vector.hpp>    //向量指针容器头文件
using namespace boost;

int main()
{
  typedef ptr_vector<string> ptr_vec;            //一个容纳 string 的指针容器
  ptr_vec vec;                                   //声明一个指针容器实例

  auto_ptr<string> ap(new string("123"));        //一个自动指针
  vec.push_back(ap);                             //指针容器可以“容纳”自动指针
  assert(ap.get() == 0);                         //自动指针失去指针的管理权

  vec.push_back(auto_ptr<string>(new string("abc")));
  vec.push_back(new string("xyz"));
  assert(vec.size() == 3);

  ptr_vec::auto_type p =                     //auto_type 是一个智能指针类型
      vec.release(vec.begin());              //指针容器释放一个迭代器位置指针的管理权
  assert(vec.size() == 2);                   //指针容器不再管理已经释放的指针
  assert(p && *p == "123");                  //p 可以像指针一样使用

  string* sp = p.release();                  //auto_type 也可以释放指针的管理权
  assert(!p);                                //p 不再管理原始指针
  delete sp;                                 //原始指针可以手动删除

  ptr_vec vec2;                              //另一个指针容器实例

  //可以在两个容器间转移指针的所有权
  vec2.transfer(vec2.end(),vec);
  assert(vec.empty());                       //转移后原指针容器不再管理指针
  assert(vec2.size() == 2);                  //新指针容器唯一管理指针
}
```

这段代码主要演示了指针容器的最基本特性：指针的所有权是唯一的，它采取的是专有（exclusive）语义而非共享语义（shared）。这意味着指针容器是指针的唯一持有者，其他人可以使用但不能管理指针。这个特性保证了指针的安全性，但同时也衍生出了许多变化，

一定程度上增加了指针容器的复杂程度[①]。

指针容器可以“容纳”标准库的自动指针 auto_ptr，它也与 auto_ptr 的语义保持一致：容纳的同时也接管了 auto_ptr 里原始指针的管理权，令 auto_ptr 不再对原始指针的生存周期负责。

虽然指针容器管理指针，但不必永远对指针负责，如果需要也可以释放指针的管理权。成员函数 release() 可以释放一个指针，它从容器中删除指针（但不删除指针指向的内容），返回一个类型为 auto_type 的智能指针，这个智能指针可以像原始指针一样使用，我们也可以从中获得原始指针。指针容器的成员函数 transfer() 可以从另一个指针容器中转移指针的所有权（参见 5.3 小节），效果上很类似元素的拷贝，但转移后原容器就失去了指针的所有权，同时目标容器获得了指针的所有权。

请读者注意：在任何时刻，这些转移操作中指针的所有权都被唯一的管理者持有，不会出现共同管理（shared）的局面。

5.1.2 指针容器的优缺点

与容纳元素的拷贝的标准容器或容纳智能指针的容器相比较，ptr_container 库有许多优点值得我们去尝试：

- 可以直接在容器中存储动态创建的对象，并且无须关心它的生存周期问题，绝对不会发生内存泄漏，用起来更安全；
- 特别支持“容纳”auto_ptr 对象，可以完美地配合 auto_ptr 工作（实际上是使用了 auto_ptr 的转移语义，在进入容器后 auto_ptr 就失去了对指针的管理权，见上一小节的示例）；
- 因为直接存储原始指针，没有了智能指针的引用计数，执行效率更高，使用的内存更少；
- 同样因为只存储原始指针，指针容器对元素的要求很低，可以存储不可拷贝构造、缺省构造和赋值的类型——这些类型是标准容器无法容纳的；

---

① 在指针管理方面 ptr_container 有些类似标准库的智能指针 std::auto_ptr，像是它的可容纳多个元素的泛化，我们可以类比来理解：
std::auto_ptr 可以看做是只能容纳一个指针的指针容器，它持有指针的所有权，而且指针的所有权是专有的而不是共享的，std::auto_ptr 的拷贝、赋值都是转移语义，操作的是指针而不是指针的指向物，一个指针同时只能被一个 std::auto_ptr 唯一管理。

- 不仅可以存储具体类，也可以存储抽象类，也就是说可以是多态的容器；
- 提供异常安全保证；
- 拥有与标准容器近似的风格，许多名字和用法都非常相似，学习成本低。

当然，指针容器并不是完美无瑕的，它并不能替代标准容器，也存在一些缺点：

- 存储的指针是专有的（exclusive），缺少智能指针容器的灵活性；
- 指针的转移、克隆（clone）等深层次概念较难理解；
- 较标准容器的接口有少量但关键的变动，使用时需要小心谨慎；
- 部分标准算法不能用于指针容器（但提供了等价的成员函数，参见 5.18 小节）。

了解了指针容器的优点和缺点，有利于我们在实际工作时针对具体问题选择最佳的解决方案。通常情况下，如果我们不得不动态创建对象，并且要使用容器来管理这些对象，那么就可以使用指针容器——但需要共享对象的所有权时除外。

5.1.3 可克隆概念

指针容器仅容纳指针，因此对元素的要求远比标准容器的要低，元素类型 T 不必是可缺省构造、可拷贝和可赋值的，几乎任何类型都可以放入指针容器。但如果容纳的对象需要创建副本，那么 T 应该是可克隆的（cloneable）——我们可以把克隆操作看做是指针容器范畴中的拷贝操作。

“克隆”（clone）是原型设计模式（prototype）[①]的具体应用，ptr_container 库使用克隆分配器（clone allocator）代替标准容器中的内存分配器概念，用来创建等价的对象。一个对象如果是可克隆的，那么它应该支持下面的两个函数：

- new_clone()　：创建一个与原型等价（equivalent）的新对象，相当于 new；
- delete_clone()：删除之前创建的对象，相当于 delete。

注意，可克隆性对于指针容器来说不是必须的（非强制），只有我们确实需要从指针容器中克隆对象时克隆函数才会被使用，大多数指针容器的操作没有对可克隆性的要求。而且为了便于库的使用，ptr_container 库在头文件 <boost/ptr_container/clone_allocator.hpp>中提供了克隆所需函数的泛型实现，代码如下：

① prototype 是一个工厂模式，可以拷贝原型实例创建一个等价的新对象，即克隆。

```
template< typename T >
inline T* new_clone( const T& r )
{
    T* res = new T( r );
    return res;
}

template< typename T >
inline T* new_clone( const T* r )
{   return r ? new_clone( *r ) : 0;}

template< typename T >
inline void delete_clone( const T* r )
{   checked_delete( r );}
```

因此，绝大多数类（通常都有缺省的拷贝构造函数）就自动支持了可克隆概念，可以使用指针容器的所有功能[①]。

关于可克隆概念的进一步讨论参见 5.19.5 小节。

5.1.4 克隆分配器

克隆分配器（clone allocator）相当于标准容器中的内存分配器（allocator）的概念和地位，是对内存模型的一种抽象表述，指针容器使用克隆分配器来克隆（不是创建）和删除指针对象。

ptr_container 库提供两个克隆分配器：heap_clone_allocator 和 view_clone_allocator。

heap_clone_allocator

heap_clone_allocator 是 ptr_container 库所有指针容器缺省使用的克隆分配器，它使用 new_clone()/delete_clone()来分配/释放内存，因此要求元素必须满足可克隆的概念。

heap_clone_allocator 的实现代码如下：

```
struct heap_clone_allocator
{
```

① 但在自己的名字空间中为自己的类提供克隆函数总是个好主意，这使得我们不会依赖于缺省的实现。

```
    template< class U >
    static U* allocate_clone( const U& r )
    {   return new_clone( r ); }

    template< class U >
    static void deallocate_clone( const U* r )
    {   delete_clone( r ); }
};
```

view_clone_allocator

view_clone_allocator 是一个“假的”克隆分配器，它并不真正管理内存，而是提供一个只读的“指针视图”，它的实现代码如下：

```
struct view_clone_allocator
{
    template< class U >
    static U* allocate_clone( const U& r )
    {   return const_cast<U*>(&r); }

    template< class U >
    static void deallocate_clone( const U* /*r*/ )
    {   /*do nothing*/    }
};
```

view_clone_allocator 并不真正的克隆或删除对象，因此如果指针容器使用它作为分配器就不会持有指针的管理权，变成了一个安全的观察者。

view_clone_allocator 可以把指针容器转化为另一个容器的视图来使用，方便我们操作，详见 5.19.4 小节。

5.1.5 指针容器的分类

与标准容器一样，指针容器也分为两大类：序列容器和关联容器，基本上标准容器都有对应的加 ptr_前缀的同名指针容器（实际上它们都是基于原标准容器实现的）。

序列指针容器

ptr_container 库目前提供的序列指针容器包括：

- ptr_vector　　　　　：类似 std::vector 的向量容器；
- ptr_deque　　　　　：类似 std::deque 的双向队列容器；
- ptr_list　　　　　　：类似 std::list 的双向链表容器；
- ptr_array　　　　　：类似 boost::array（或 std::tr1::array）的数组容器；
- ptr_circular_buffer：类似 boost::circular_buffer 的循环缓冲区容器。

序列指针容器类的关系如图 5-1 所示：

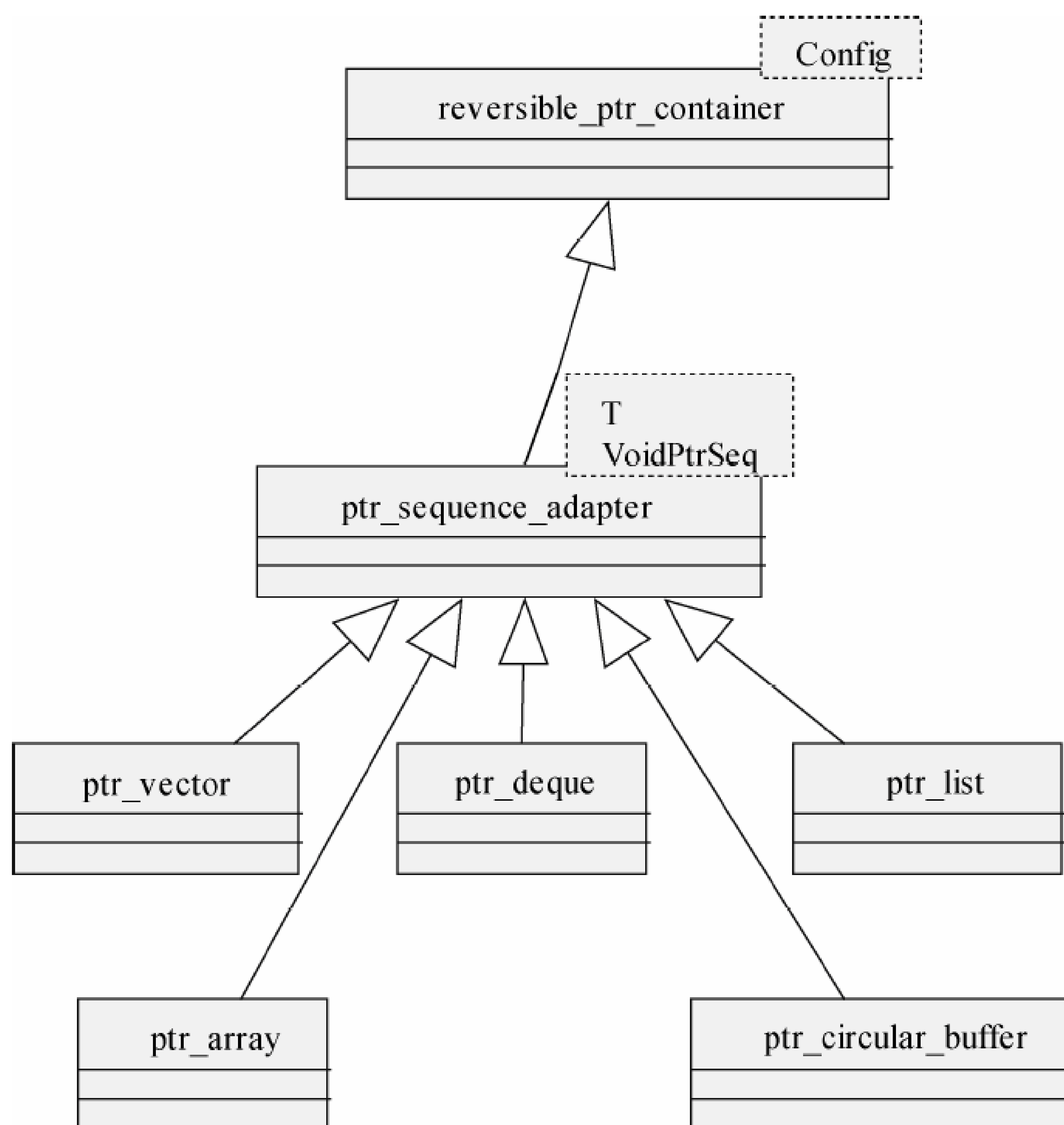


图 5-1　序列指针容器关系图

关联指针容器

ptr_container 库目前提供的关联指针容器包括：

- ptr_set　　　：类似 std::set 的有序集合容器；
- ptr_multiset：类似 std::multiset 的有序集合容器；

- ptr_map　　　　　　：类似 std::map 的有序映射容器；
- ptr_multimap　　　：类似 std::multimap 的有序映射容器；
- ptr_unordered_set：类似 boost::unordered_set（或 std::tr1::unordered_set）的无序集合容器；
- ptr_unordered_map：类似 boost::unordered_map（或 std::tr1::unordered_map）的无序映射容器。

关联指针容器类的关系如图 5-2 所示：

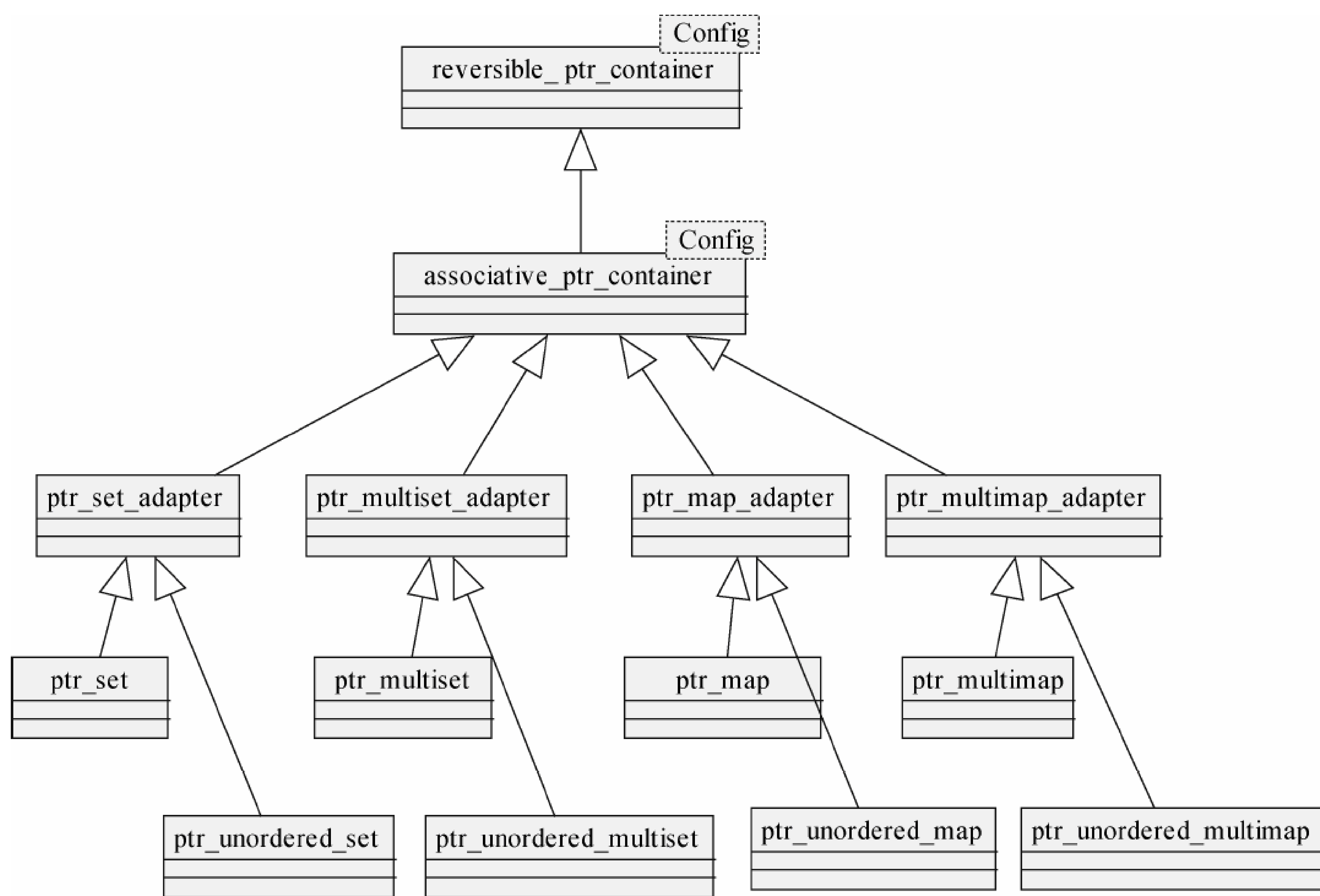


图 5-2　关联指针容器关系图

由于 ptr_container 库的内容几乎与标准库的容器同样大，所以本书不能完全介绍所有相关的接口和用法，与标准容器基本相同或者不太重要的内容会一带而过，请读者见谅。

本章接下来的部分做如下安排：以类图为脉络先研究指针容器的共通能力，然后顺序研究序列指针容器和关联指针容器，再研究指针容器相关的空指针、算法等深层次概念。

5.2 指针容器的共通能力

reversible_ptr_container 是 ptr_container 库中所有指针容器的基类，包含了指针容器的基本操作，它位于名字空间 boost::ptr_container_detail，通过研究它可以了解指针容器的基本实现原理，加深对指针容器的认识。

reversible_ptr_container 的接口很多，为了使叙述清晰，它的类摘要将逐步介绍。

5.2.1 模板参数

reversible_ptr_container 模板参数相关的代码摘要如下：

```
template<class Config, CloneAllocator>
class reversible_ptr_container
{
private:
    typedef Config::value_type Ty_;                    //指针容器的内部值类型
    typedef Config::void_container_type  Cont;         //指针容器的内部容器类型
public:
    Cont&       base();
public:
    //基本类型定义
    typedef  Ty_*          value_type;                 //容器值类型，即指针
    typedef  Ty_*          pointer;                    //容器指针类型，同值类型
    typedef  Ty_&          reference;                  //容器的值引用类型
    typedef  const Ty_&    const_reference;            //容器的 const 值引用类型

   //迭代器类型
    typedef  Config::iterator  iterator;
    typedef  boost::reverse_iterator< iterator >  reverse_iterator;

   //指针容器内部的智能指针类型
    typedef  static_move_ptr<Ty_,Deleter> auto_type;
};
```

reversible_ptr_container 主要的模板参数是 Config，它实际上是一个可返回多个元数据的非标准元函数，用于元计算有关容器的各种类型，如值、迭代器等。Config 的摘要如下：

```
struct Config
{
    typedef some_define    void_container_type;        //容器类型
    typedef some_define    allocator_type;             //内存分配器类型
    typedef some_define    value_type;                 //值类型
    typedef some_define    iterator;                   //迭代器类型
    typedef some_define    const_iterator;             //const 迭代器类型
};
```

reversible_ptr_container 使用 Config 元函数大大简化了指针容器所需的模板数量，可以一次性的获得多个构造指针容器所必需的类型，对于 reversible_ptr_container 来说最重要的就是值类型 Config::value_type 和迭代器类型 Config::iterator/const_iterator，它们定义了指针容器存储的值类型、引用类型和迭代器类型，具体的定义则因具体实现而不同。

ptr_container 内部使用 void* 来存储指针[①]，类型 Config::void_container_type 定义了容纳 void*指针的容器，是各种与标准容器对应的指针容器的内部实现，它被重定义为 Cont 类型，成员函数 base()可以获得这个内部容器的引用。

reversible_ptr_container 里另一个重要的类型是 auto_type。它的真实类型是 ptr_container_detail::static_move_ptr<>，是一个类似 std::auto_ptr 的智能指针，使用 boost::compressed_pair（见 2.1 小节）来存储指针和对应的删除器，支持隐式 bool 检查，析构时会自动删除指针，release()成员函数同样不会删除指针，仅仅是释放所有权。

下面的代码使用 type_traits 库验证了 ptr_vector 容器的一些内部类型定义：

```
typedef ptr_vector<string> ptr_vec;

assert((is_same<ptr_vec::value_type, string*>::value));
assert((is_same<ptr_vec::pointer    , string*>::value));
assert((is_same<ptr_vec::reference , string&>::value));
```

① 据本库的作者称将来 ptr_container 可能会转换到 T*的实现，但目前尚未有所动作。

5.2.2 构造与赋值

reversible_ptr_container 支持多种形式的构造函数和赋值操作，下面的代码仅列出了其中较常用的一部分：

```
class reversible_ptr_container
{
public:
    //构造函数
    reversible_ptr_container();

    reversible_ptr_container( const reversible_ptr_container& r );
    template< class C, class V >
    reversible_ptr_container( const reversible_ptr_container<C,V>& r );
    template< class InputIterator >
    reversible_ptr_container( InputIterator first, InputIterator last);

    template< class PtrContainer >
    explicit reversible_ptr_container( std::auto_ptr<PtrContainer> clone ) ;

    //析构函数
    ~reversible_ptr_container();

    //赋值操作
    reversible_ptr_container& operator=( const reversible_ptr_container& r );
    template< class PtrContainer >
    reversible_ptr_container& operator=( std::auto_ptr<PtrContainer> clone );
};
```

reversible_ptr_container 有四种形式的构造函数：

- 最简单的缺省构造函数可以创建出一个不包含任何指针的空容器；
- 如果传递另一个指针容器或者一个迭代器区间，那么构造函数会使用克隆分配器克隆其中的所有元素，新容器拥有原容器里所有元素的等价副本，但原容器里的指针不受影响；
- 如果把一个指针容器的自动指针传递给构造函数，那么指针容器将使用转移语义，接管原容器的所有指针。

reversible_ptr_container 的赋值操作与同参数的构造函数的功能和效果类似，但它只有两种形式，分别支持从一个指针容器或者一个自动指针赋值。

示范这些构造函数和赋值操作的代码如下：

```
int main()
{
  typedef ptr_vector<string> ptr_vec;     //一个容纳 string 的指针容器

  ptr_vec vec;                            //无参缺省构造函数
  assert(vec.empty());                    //指针容器为空

  vec.push_back(new string("123"));       //添加两个元素
  vec.push_back(new string("abc"));
  assert(vec.size() == 2);

  ptr_vec vec2(vec);                      //从另一个容器克隆构造
  assert(vec2.size() == 2);               //两个容器各自拥有等价的对象
  assert(vec.size() == 2);

  auto_ptr<ptr_vec> apv = vec.release(); //释放所有指针的所有权
  assert(vec.empty());

  ptr_vec vec3(apv);                      //从一个自动指针构造
  assert(vec3.size() == 2);

  ptr_vec vec4, vec5;                     //两个指针容器缺省构造

  vec4 = vec2;                            //赋值操作，克隆
  assert(vec2.size() == 2);               //两个容器各自拥有等价的对象
  assert(vec4.size() == 2);

  vec5 = vec3.release();                  //从一个自动指针赋值
  assert(vec3.empty());
  assert(vec5.size() == 2);
}
```

5.2.3 访问元素

reversible_ptr_container 提供了一些通用的访问元素的接口，代码摘要如下：

```
class reversible_ptr_container
{
public:
    //插入操作
    iterator insert( iterator before, Ty_* x );
    template< class U >
    iterator insert( iterator before, std::auto_ptr<U> x );

    //删除操作
    iterator erase( iterator x ) ;
    iterator erase( iterator first, iterator last ) ;

    //替换操作
    auto_type replace( iterator where, Ty_* x ) ;
    template< class U >
    auto_type replace( iterator where, std::auto_ptr<U> x );

    //释放所有权操作
    auto_type release( iterator where );
    std::auto_ptr<this_type> release();

    //克隆操作
    std::auto_ptr<this_type> clone() const;
}
```

reversible_ptr_container 的插入、删除操作与标准容器相同，只是多了对 auto_ptr 的重载：都是对当前位置进行操作，并返回操作后的新位置。

replace()是指针容器独有的能力，它可以在容器中替换当前位置的指针，然后以 auto_type 返回当前位置的原指针，有点类似于赋值操作。

成员函数 release()可以释放一个指针或者整个指针容器的所有权，这可以在函数返回时使用，避免了昂贵的拷贝操作。成员函数 clone()同样返回一个持有指针容器的自动指针，但它使用了克隆构造函数，返回一个与本容器等价的克隆副本，并不释放指针的所有权。

示范这些操作的代码如下：

```
int main()
{
  typedef ptr_vector<string> ptr_vec;    //一个容纳 string 的指针容器
  ptr_vec vec;
```

```
    vec.push_back(new string("123"));  //添加两个元素
    vec.push_back(new string("abc"));
    assert(vec.size() == 2);

    ptr_vec::iterator pos =            //在容器前端执行插入操作，返回新元素的位置
        vec.insert(vec.begin(), new string("000"));
    assert(vec.size() == 3);
    assert(pos == vec.begin());

    ++pos;                             //pos 指向元素“123”
    pos = vec.erase(pos);              //删除操作，返回下一个元素的位置
    assert(vec.size() == 2);
    assert(*pos == "abc");

    ptr_vec::auto_type p =             //替换操作，返回被替换的元素
        vec.replace(pos, new string("xyz"));
    assert(*pos == "abc");
    assert(vec.size() == 2);

    ptr_vec vec2;
    vec2 = vec.clone();                //克隆指针容器并赋值，与 release()不同
    assert(vec.size() == 2);           //原容器元素不变
    assert(vec2.size() == 2);
}
```

5.2.4 其他能力

除了以上列举的能力之外，reversible_ptr_container 还具有标准容器所应当具有的一些接口，包括容量、迭代器等操作。因为它们的功能都很简单，而且大都是直接委托给内部的容器处理，所以下面仅列出接口的声明，不再做解释。值得注意的是指针容器的比较操作是基于指针指向的内容，而不是直接比较指针，这是与标准容器一致的。

reversible_ptr_container 的其他接口代码摘要如下：

```
class reversible_ptr_container
{
public:
    //容量操作
    size_type size() const;
```

```
    size_type max_size() const;
    bool empty() const;
    void swap( reversible_ptr_container& r );
    void clear():

public:
    //迭代器操作
    iterator begin();
    iterator end();
    reverse_iterator rbegin();
    reverse_iterator rend();

    const_iterator cbegin() const;
    const_iterator cend() const;
    const_reverse_iterator crbegin() const;
    const_reverse_iterator crend() const;

 public:
    //比较操作
    ...//各种比较操作符的重载，包括==、!=、<等
};
```

5.3 序列指针容器适配器

ptr_sequence_adapter 是序列指针容器的基类，它使用适配器设计模式把序列普通容器适配成序列指针容器，适配的对象有 std::vector、std::list、boost::array 等，如果有必要，也可以让它适配其他容器实现定制自定义的指针容器。ptr_sequence_adapter 位于头文件 <boost/ptr_container/ptr_sequence_adapter.hpp>。

5.3.1 配置元函数

ptr_sequence_adapter 使用的配置元函数是 ptr_container_detail::sequence_config，它的类摘要如下：

```
template< class T, class VoidPtrSeq>
struct sequence_config
{
```

```
    typedef  remove_nullable<T>::type U;                    //元素值类型
    typedef  VoidPtrSeq void_container_type;                //被适配的容器类型
    typedef  VoidPtrSeq::allocator_type allocator_type;     //内存分配器
    typedef  U   value_type;                                //值类型
    typedef  void_ptr_iterator<VoidPtrSeq::iterator, U >
             iterator;                                      //迭代器类型
    typedef  void_ptr_iterator<VoidPtrSeq::const_iterator, const U >
             const_iterator;                                //const 迭代器类型
};
```

sequence_config 有两个模板参数，T 是容器的元素类型，VoidPtrSeq 是容纳 void*指针的序列容器类型，元函数 remove_nullable<T>用于移除元素类型可为空指针的修饰（参见 5.9.2 小节），得到真正的元素类型 U。

sequence_config 中的 void_ptr_iterator 是一个完整的迭代器类，它适配了操作 void*指针元素的迭代器 VoidPtrSeq::iterator，使用 iterator_traits（参见 3.3 节）元计算与容器迭代器相关的迭代器类型，在 operator*()、operator->()等操作符重载中用 static_cast<>把 void*指针转换成 T*，使得我们可以方便地用迭代器直接操作元素而非指针。

5.3.2 类摘要

ptr_sequence_adapter 是 reversible_ptr_container 的子类，具有 reversible_ptr_container 的全部能力，故下面的类摘要仅列出新增的接口：

```
template
< class T, class VoidPtrSeq,
   class CloneAllocator = heap_clone_allocator
>
class ptr_sequence_adapter : public
    reversible_ptr_container<
         sequence_config<T,VoidPtrSeq>, CloneAllocator >
{
public:
    //构造与赋值
    ... //同 reversible_ptr_container
    template< class InputIterator >
    assign( InputIterator first, InputIterator last );
```

```
public:
    //访问元素
    T& front();
    T& back();

    void push_back( T* x );
    template< class U >
    void push_back( std::auto_ptr<U> x );
    auto_type pop_back();

    void resize( size_type size );
    void resize( size_type size, T* to_clone );

public:
    //指针所有权转移
    template< class PtrSequence >
    void transfer( iterator before,
         PtrSequence::iterator object, PtrSequence& from );

    template< class PtrSequence >
    void transfer( iterator before,
          PtrSequence::iterator first, PtrSequence::iterator last,
          PtrSequence& from );

    template< class PtrSequence >
    void transfer( iterator before, PtrSequence& from );

    void transfer( iterator before, value_type* from,
                 size_type size, bool delete_from = true );
};
```

ptr_sequence_adapter 有三个模板参数，前两个（T 和 VoidPtrSeq）被传递给 sequence_config 形成配置元函数，再和克隆分配器参数 CloneAllocator 一起传递给 reversible_ptr_container。

ptr_sequence_adapter 还提供了一些内置的算法，如 sort()、erase_if()、merge()和 unique()等，这些将在 5.18 小节讨论。

5.3.3　接口解说

ptr_sequence_adapter 具有序列容器的通用接口，如 assign()、front()、push_back()，这些接口的功能与标准容器的同名接口完全相同，用法也是一样的。在这里我们要稍微留意一下 resize()函数：如果扩大容器的容量时它需要使用 T 的缺省构造函数来创建对象填充序列，或者是使用一个指定的克隆。

transfer()系列函数用于指针的所有权管理，是指针容器的核心操作，它们可以在两个指针容器之间移动指针，具体效果如下：

- transfer(before,object,from)：把 object 插入到当前容器的 before 位置之前，并从 from 中移除所有权。
- transfer(before,first,last,from)：把[first,last)之间的元素插入到当前容器的 before 位置之前，并从 from 中移除这些元素的所有权。
- transfer(before,from)　　：把 from 里的所有元素插入到当前容器的 before 位置之前，之后 from 不再持有任何元素。

5.3.4　代码示例

我们仍以最简单的 ptr_vector 来示范这些序列指针容器的共通能力，首先是基本的元素访问功能：

```
int main()
{
  typedef ptr_vector<string> ptr_vec;          //容纳 string 的指针容器
  ptr_vec vec;

  vec.push_back(new string("123"));            //添加两个元素
  vec.push_back(new string("abc"));

  assert(vec.front() == "123");                //使用 front()获取序列前端元素
  assert(vec.back() == "abc");                 //使用 back()获取序列末端元素

  ptr_vec vec2;
  vec2.assign(vec.begin(), vec.end());         //从另一个指针容器赋值
```

```
    assert(vec2.size() == 2);

    assert(*vec2.pop_back() == "abc");      //弹出序列末端元素
    assert(vec2.size() == 1);

    vec.resize(5);                          //修改容器的大小，多出的元素使用缺省构造
    assert(vec.back().empty());

    vec.resize(10, new string("xyz"));      //修改容器的大小，多出的元素使用克隆
    assert(vec.back() == "xyz");
}
```

接下来我们使用 transfer() 系列函数来转移指针的所有权：

```
int main()
{
    ...     //同前

    //转移一个元素
    vec.transfer(vec.end(), vec2.begin(), vec2);
    assert(vec.size() == 11);
    assert(vec2.size() == 0);

    //转移一个迭代器区间
    vec2.transfer(vec2.end(), vec.begin(),vec.begin() + 5, vec);
    assert(vec.size() == 6);
    assert(vec2.size() == 5);

    //转移整个容器
    vec.transfer(vec.begin(), vec2);
    assert(vec.size() == 11);
    assert(vec2.size() == 0);
}
```

5.4 ptr_vector

指针容器 ptr_vector 是最简单常用的指针容器，它基于标准容器 std::vector 容纳 void*指针，使用 ptr_sequence_adapter 适配实现，位于头文件<boost/ptr_container/ptr_vector.hpp>。

5.4.1 类摘要

ptr_vector 是 ptr_sequence_adapter 的子类，下面的类摘要仅列出它的新增接口：

```
template
<  class T,
   class CloneAllocator = heap_clone_allocator,
   class Allocator      = std::allocator<void*> >
class ptr_vector : public ptr_sequence_adapter
                      <  T,
                         std::vector<void*,Allocator>,
                         CloneAllocator>
{
public:
    //构造函数
    explicit ptr_vector( size_type n );

public:
    //容量
    size_type  capacity() const;
    void       reserve( size_type n );

public:
    //访问元素
    T&         operator[]( size_type n );
    T&         at( size_type n );

public:
    //替换操作
    auto_type replace( size_type idx, T* x );
    template< class U >
    auto_type replace( size_type idx, std::auto_ptr<U> x );

public:
    //C 风格数组的支持
    void transfer( iterator before, T** from,
                   size_type size, bool delete_from = true );
    T**  c_array();
};
```

ptr_vector 有三个模板参数，但通常只给定一个元素类型 T 即可，它会自动把 T 和 std::vector<void*>提供给父类 ptr_sequence_adapter 完成模板参数的配置。

ptr_vector 基于标准容器 std::vector，接口与 std::vector 基本相同，大部分操作都使用 base()转发给内部的容器实现，因此很容易理解。但构造函数 ptr_vector(n) 的行为不同于 std::vector，它不会创建 n 个元素，而是保留 n 个元素的空间，相当于调用 reserve(n)。

因为 ptr_vector 支持随机访问，所以它有 operator[]，可以用整数索引直接访问元素，也可以用索引来指示位置来替换元素。

为了像 std::vector 一样提供对数组的良好兼容性，ptr_vector 增加了一个 c_array()函数，它可以返回一个类型为 T**的指针，即 T*[]，可以传递给 C 风格的 API。成员函数 transfer()也有对应的数组形式的重载，可操作动态创建的指针数组，把大小为 size 的 from 数组中的所有元素插入到当前容器的 before 位置之前，如果 delete_from == true，那么函数执行后 from 将被删除（即 delete[] from）。

5.4.2 用法

ptr_vector 继承了 std::vector 的优良传统，无疑是所有指针容器中最容易使用的一个，之前已经使用它多次演示了指针容器的用法，现在只需要很少的代码来展示它其余的特性：

```
#include <boost/ptr_container/ptr_vector.hpp>    //向量指针容器头文件
using namespace boost;
int main()
{
  typedef ptr_vector<string> ptr_vec;            //向量指针容器
  ptr_vec vec(10);                               //保留 10 个元素的空间待用

  assert(vec.empty());                           //此时容器内无元素
  assert(vec.capacity() == 10);                  //可容纳 10 个元素

  vec.push_back(new string("star"));             //尾部添加一个元素
  assert(vec[0] == "star");

  vec.replace(0, new string("fox"));             //使用整数索引
  assert(vec[0] == "fox");                       //使用 operator[]
```

```
    string** arr = new string*[2];                    //一个动态指针数组
    arr[0] = new string("123");
    arr[1] = new string("abc");

    vec.transfer(vec.begin(), arr, 2);                //转移指针的所有权
    assert(vec.size() == 3);                          //此时原动态指针数组已经失效

    string** p = vec.c_array();                       //获得指针数组
    assert(*p[0] == "123" && *p[2] == "fox");
}
```

5.5 ptr_deque

指针容器 ptr_deque 是一个双向队列指针容器，它基于标准容器 std::deque 容纳 void*指针，使用 ptr_sequence_adapter 适配实现，位于头文件<boost/ptr_container/ptr_deque.hpp>。

5.5.1 类摘要

ptr_deque 是 ptr_sequence_adapter 的子类，类摘要如下：

```
template
<  class T,
   class CloneAllocator = heap_clone_allocator,
   class Allocator      = std::allocator<void*> >
class ptr_deque : public ptr_sequence_adapter
                  <  T,
                     std::deque<void*,Allocator>,
                     CloneAllocator>
{
public:
    //访问元素
    T&        operator[]( size_type n );
    T&        at( size_type n );

    void      push_front( T* x );
```

```
    template< class U >
    void     push_front( std::auto_ptr<U> x );
    auto_type pop_front();

public:
    //替换操作
    auto_type replace( size_type idx, T* x );
    template< class U >
    auto_type replace( size_type idx, std::auto_ptr<U> x );

public:
    //C 风格数组的支持
    void transfer( iterator before, T** from,
                   size_type size, bool delete_from = true );
    T**  c_array();
};
```

ptr_deque 的接口非常类似 ptr_vector，但因为它是可以双向增长的，所以没有 capacity()和 reserve()这样的容量相关操作，并且增加了可以在前端处理元素的 push_front()和 pop_front()函数，这些变化与标准容器 deque 是一致的。

5.5.2 用法

ptr_deque 的用法与 ptr_vector 几乎是相同的，下面的代码与 5.4.2 小节的代码仅有少量的差异，读者可对比阅读。

```
#include <boost/ptr_container/ptr_deque.hpp>
using namespace boost;

int main()
{
  typedef ptr_deque<string> ptr_dq;          //双向队列指针容器
  ptr_dq dq;                                 //队列指针容器无须设置容量

  assert(dq.empty());

  dq.push_front(new string("mario"));        //直接前端插入元素
  dq.push_back(new string("peach"));         //也可以后端直接插入元素
  assert(dq[0] == "mario");                  //使用 operator[]
```

```
    dq.replace(0, new string("luigi"));         //同样可以使用整数索引操作
    assert(dq.front() == "luigi");

    string** arr = new string*[2];              //一个动态指针数组
    arr[0] = new string("123");
    arr[1] = new string("abc");

    dq.transfer(dq.begin(), arr, 2);            //转移指针的所有权
    assert(dq.size() == 4);

    string** p = dq.c_array();                  //获得指针数组
    assert(*p[0] == "123" && *p[2] == "luigi");
}
```

5.6 ptr_list

指针容器 ptr_list 是一个双向链表指针容器，它基于标准容器 std::list 容纳 void*指针，使用 ptr_sequence_adapter 适配实现，位于头文件<boost/ptr_container/ptr_list.hpp>。

5.6.1 类摘要

ptr_list 是 ptr_sequence_adapter 的子类，类摘要如下：

```
template
<   class T,
    class CloneAllocator = heap_clone_allocator,
    class Allocator      = std::allocator<void*> >
class ptr_list : public
    ptr_sequence_adapter< T,
                          std::list<void*,Allocator>,
                          CloneAllocator >
{
public:
    //访问元素
    void      push_front( T* x );
    template< class U >
    void      push_front( std::auto_ptr<U> x );
```

```
    auto_type pop_front();
public:
    //C 风格数组的支持
    void transfer( iterator before, T** from,
                   size_type size, bool delete_from = true );
};
```

Ptr1_list 使用 std::list 作为内部容器，因而具有与 std::list 相同的性质，它使用双向链表存储元素，不能随机访问，不提供 operator[]，可以双向迭代，可以在任意的位置插入或删除元素。

同标准容器 std::list 一样，ptr_list 也有一些特殊的算法成员函数，这些将在 5.18 小节描述。

5.6.2 用法

下面的代码简单示范了 ptr_list 的用法，与 ptr_vector、ptr_deque 类似：

```
#include <boost/ptr_container/ptr_deque.hpp>
#include <boost/ptr_container/ptr_list.hpp>
using namespace boost;

int main()
{
  typedef ptr_list<string> ptr_lt;           //双向链表指针容器
  ptr_lt lt;

  lt.push_front(new string("mario"));        //直接前端插入元素
  lt.push_back(new string("peach"));         //直接后端插入元素
  lt.insert(lt.end(), new string("yoshi"));  //序列中间任意位置插入元素
  assert(lt.size() == 3);

  //使用迭代器区间向另一个指针容器传递克隆
  ptr_deque<string> dq(lt.begin(), boost::next(lt.begin(), 2));
  assert(dq.size() == 2);
  assert(lt.size() == 3);
}
```

5.7 ptr_array

指针容器 ptr_array 是一个不可动态增长的静态数组指针容器，它基于容器 boost::array 容纳 void*指针，使用 ptr_sequence_adapter 适配实现，位于头文件 <boost/ptr_container/ptr_array.hpp>。

5.7.1 类摘要

ptr_array 是 ptr_sequence_adapter 的子类，但因为它的内部容器 boost::array 不符合标准容器的定义且大小固定，故其实现和接口较 ptr_vector、ptr_deque 等有所不同。它的类摘要如下：

```
template
<
    class T,
    size_t N,
    CloneAllocator = heap_clone_allocator
>
class ptr_array : public
    ptr_sequence_adapter< T,
                         ptr_array <void*,N>,
                         CloneAllocator >
{
public:
    //构造和赋值
    ptr_array();
    explicit ptr_array( const ptr_array& r );
    template< class U >
    explicit ptr_array( const ptr_array<U,N>& r );
    explicit ptr_array( std::auto_ptr<ptr_array>& r );

    ptr_array& operator=( const ptr_array& r );
    template< class U >
    ptr_array& operator=( const ptr_array<U,N>& r );
    ptr_array& operator=( std::auto_ptr<this_type> r );

private:
    //禁用序列变动操作
```

```
    using base_class::insert;
    using base_class::erase;
    using base_class::push_back;
    using base_class::push_front;
    using base_class::pop_front;
    using base_class::pop_back;
    using base_class::transfer;

public:
    //访问元素
    T&        front();
    T&        back();

    T&        at( size_t );
    template< size_t idx >
    T&        at();

    T&        operator[]( size_t );

public:
    //替换操作
    template< size_t idx >
    auto_type replace( T* r );
    template< size_t idx, class U >
    auto_type replace( std::auto_ptr<U> r );

    auto_type replace( size_t idx, T* r );
    template< class U >
    auto_type replace( size_t idx, std::auto_ptr<U> r );

public:
    //指针所有权转移
    std::auto_ptr<ptr_array>  clone() const;
    std::auto_ptr<ptr_array>  release();
};
```

5.7.2 用法

ptr_array 与 boost::array 很像，它是一个大小固定的容器，所以缺省构造函数将使用空指针填满容器，这是与其他序列指针容器相比一个显著不同的地方（其他序列指针容

器缺省构造出一个空容器)。

因为 ptr_array 不能改变大小，所以它私有化了 insert()、erase()等操作，但仍然可以像 ptr_vector 一样使用整数索引来访问元素。值得注意的是 ptr_array 特别提供了编译期访问元素的函数，可以用模板参数指定元素的索引，有利于提高运行效率。

示范 ptr_array 用法的代码如下：

```
#include <boost/ptr_container/ptr_array.hpp>
using namespace boost;

int main()
{
  typedef ptr_array<string, 5> ptr_arr;              //大小为 5 的数组指针容器
  ptr_arr arr;

  assert(!arr.empty());                              //容器不空
  assert(arr.size() == 5 && arr.max_size() == 5);    //固定长度是 5
  assert(arr.base().front() == NULL);                //所有元素均为空指针

  arr.replace(0, new string("metroid"));             //使用 replace()加入元素
  arr.replace<2>(new string("prime"));               //编译期 replace()用法
  arr.replace<4>(new string("other m"));
  assert(arr.front() == "metroid");                  //使用 front()访问前端
  assert(arr.back() == "other m");                   //使用 back()访问后端

  arr[0] = "samus";                                  //使用 operator[]
  arr.at<2>() = "alan";                              //使用编译期 at()

  assert(*boost::next(arr.begin(), 2) == "alan");    //仍然可以使用迭代器
}
```

因为没有 push_back()、push_front()等接口，在使用 ptr_array 时我们必须使用 replace()向容器添加 new 出来的动态对象，特别要注意的是在 replace()添加元素前不能写如下的代码：

```
arr[1] = "null ptr error";                 //空指针赋值错误
```

这是因为 ptr_array 不是一个空的容器，最初里面存储的都是空指针，operator[]无法使用空指针返回一个 T&执行赋值操作，父类 ptr_sequence_adapter 在运行时会用

BOOST_ASSERT 抛出异常。

5.8 ptr_circular_buffer

指针容器 ptr_circular_buffer 是一个循环缓冲区指针容器，它基于容器 boost::circular_buffer 容纳 void*指针，使用 ptr_sequence_adapter 适配实现，位于头文件<boost/ptr_container/ptr_circular_buffer.hpp>。

5.8.1 类摘要

ptr_circular_buffer 是 ptr_sequence_adapter 的子类，接口与 boost::circular_buffer 基本一致。它的类摘要如下：

```
template
<
    class T,
    class CloneAllocator = heap_clone_allocator,
    class Allocator      = std::allocator<void*>
>
class ptr_circular_buffer : public
    ptr_sequence_adapter< T,
                          boost::circular_buffer<void*,Allocator>,
                          CloneAllocator >
{
public:
    //容量
    bool full() const;
    size_type  capacity() const;
    void       reserve( size_type n );

public:
    //访问元素
    T&         operator[]( size_type n );
    T&         at( size_type n );

    void       push_front( T* x );
    template< class U >
```

```
    void     push_front( std::auto_ptr<U> x );
    auto_type pop_front();

public:
    //C 风格数组的支持
   void transfer( iterator before, T** from,
                  size_type size, bool delete_from = true );
   T**  c_array();

public:
    //ptr_circular_buffer 特有操作
    array_range array_one();
    array_range array_two();
    pointer linearize();
    void rotate( const_iterator new_begin );
};
```

5.8.2 用法

ptr_circular_buffer 内部实现为一个有限循环队列，像 ptr_deque 一样可以在两端操作，但内部容量是循环使用的，用起来与 boost::circular_buffer 几乎相同。

示范 ptr_circular_buffer 用法的代码如下：

```
#include <boost/ptr_container/ptr_circular_buffer.hpp>
using namespace boost;

int main()
{
  typedef ptr_circular_buffer<string> ptr_buffer;  //循环缓冲区指针容器
  ptr_buffer cb(5);                                //大小为 5 的循环缓冲区

  assert(cb.empty());                              //缺省构造容器为空

  cb.push_front(new string("link"));               //使用 push_front()操作前端
  cb.push_back(new string("zelda"));               //使用 push_back()操作后端
  cb.push_back(new string("epona"));

  assert(cb.size() == 3);
  assert(!cb.full());                              //缓冲区未满
```

```
    assert(cb[1] == "zelda");                  //可以使用 operator[]

    string** p = cb.linearize();               //把循环缓冲区变为线性数组
    assert(*p[0] == "link");

    cb.rotate( boost::prior(cb.end()) );       //选中循环缓冲区
    assert(*cb.c_array()[0] == "epona");       //使用 c_array()函数，等价于线性化
}
```

5.9 空指针处理

提到指针，有一个特别的概念不得不涉及——那就是空指针（null pointer），对于专门容纳指针的 ptr_container 库来说更是一个重要的议题。

5.9.1 禁用空指针

如果不做什么特别的声明，那么 ptr_container 库的所有指针容器均不允许处理空指针，也就是说不可能向容器插入或者从容器中取出空指针（ptr_array 是一个例外）。如果向容器插入一个空指针，那么会得到一个 boost::bad_pointer 异常，比如：

```
ptr_vector<int> vec;
vec.push_back(NULL);                       //抛出 bad_pointer 异常

ptr_list<string> lt;
lt.push_front(NULL);                       //抛出 bad_pointer 异常
```

指针容器的这种处理方式很好地防止了无意中使用空指针可能带来的灾难性后果，可以让代码更加安全。

5.9.2 允许空指针

然而有的时候我们又确实需要向容器中插入空指针，ptr_container 库为此引入了一个特别的包装类：nullable<T>。

包装类 nullable<T>的定义非常简单，就是一个返回 T 的元函数：

```
template< class T >
```

```
struct nullable
{
    typedef T type;                //返回类型 T
};
```

元函数 is_nullable<T>用于检测 T 是否是一个可空指针化的类型，实现如下：

```
namespace ptr_container_detail
{
    template< class T >
    type_traits::yes_type is_nullable( const nullable<T>* );
    type_traits::no_type is_nullable( ... );
}
template< class T >
struct is_nullable
{
private:
   BOOST_STATIC_CONSTANT( T*, var );
public:
   BOOST_STATIC_CONSTANT(bool,
   value = sizeof( ptr_container_detail::is_nullable( var ) )
              == sizeof( type_traits::yes_type ) );
};
```

在名字空间 boost::ptr_container_detail 里有两个用于元计算的重载函数 is_nullable()（所以无须函数体），对 nullable<T>的重载形式返回 yes_type，其他类型则返回 no_type。is_nullable<T>使用 sizeof 操作符对函数运算，返回函数返回类型的大小，最终判定 T 是否是一个可空指针化的类。

元函数 remove_nullable<T>使用了 mpl::eval_if<>，可以移除 nullable<>的包装：

```
template< class T >
struct remove_nullable
{
    typedef  mpl::eval_if< is_nullable<T>, T,
                               mpl::identity<T> >::type
        type;
};
```

这些元函数被 reversible_ptr_container 等指针容器内部所使用，用于空指针相关的操作，感兴趣的读者可自行阅读它们的内部实现代码。

5.9.3 使用空指针

在指针容器声明时使用 nullable<T>作为元素类型（而不是直接使用 T）即表明容器可容纳空指针，例如：

```
ptr_vector<nullable<int> > vec;          //使用 nullable<T>包装元素类型
vec.push_back(NULL);                     //可插入空指针

ptr_list<nullable<string> > lt;          //使用 nullable<T>包装元素类型
lt.push_front(NULL);                     //可插入空指针
```

nullable<T>对元素类型的包装并不影响指针容器的接口，因为指针容器内部已经使用元函数 remove_nullable<T>去除了 nullable<T>的包装，它的作用仅仅是在编译期给指针容器一个提示而已，我们千万不可写出下面的代码：

```
ptr_vector<nullable<int> > vec;
vec.push_back(new nullable<int>(10));    //编译错误!!!
```

引入空指针后指针容器的处理就变得复杂了一些，因为对空指针的操作是无效的，所以在访问元素时必须时刻检查指针是否为空。ptr_container 库提供了一个自由函数 is_null()用来检查指针容器的迭代器是否指向了一个“真正的”空指针：

```
template< class Iterator, class T >
inline bool is_null( void_ptr_iterator<Iterator,T> i );
```

对于 ptr_vector、ptr_deque、ptr_array 和 ptr_circular_buffer 这样的支持整数索引的指针容器还有一个成员函数 is_null()，它也可以检查当前位置上是否是空指针：

```
bool is_null( size_t idx ) const;
```

空指针检查函数可以这样使用：

```
typedef ptr_vector<nullable<int> > ptr_null_vec;   //可容纳空指针的容器
ptr_null_vec vec;

vec.push_back(NULL);                               //添加一个空指针
```

```
    vec.push_back(new int(100));                     //添加一个正常元素

    assert(vec.is_null(0));                          //使用成员函数检查空指针
    assert(!vec.is_null(1));

    for (ptr_null_vec::iterator i = vec.begin();     //使用迭代器迭代元素
        i != vec.end(); ++i)                     //因为空指针的原因不能使用 BOOST_FOREACH
    {
        if (!boost::is_null(i))                  //使用自由函数检查是否是空指针
        {   cout << *i << " ";  }
    }
```

5.9.4　空对象模式

虽然 ptr_container 库使用 nullable<T>允许我们在容器中存储空指针，但这并不是问题的最佳解决方案，空指针会迫使我们每次访问元素时都进行检查以防止出错，破坏了代码的优雅和整洁，同时空指针的隐患并没有消除，一旦疏忽大意，对空指针的操作就会使整个程序立即崩溃。

这个时候可以采用“智能空指针”——空对象模式（null object）来解决这个问题。空对象是一个模仿了空指针的对象，它给空指针赋予了一个合理的、可接受的行为（通常是空操作），使得代码可以一致地处理实对象和空对象，无须再使用专门的条件判断语句来检查空指针。

下面首先实现一个简单的多态类继承体系，注意其中使用了 boost::noncopyable，类关系图如图 5-3 所示：

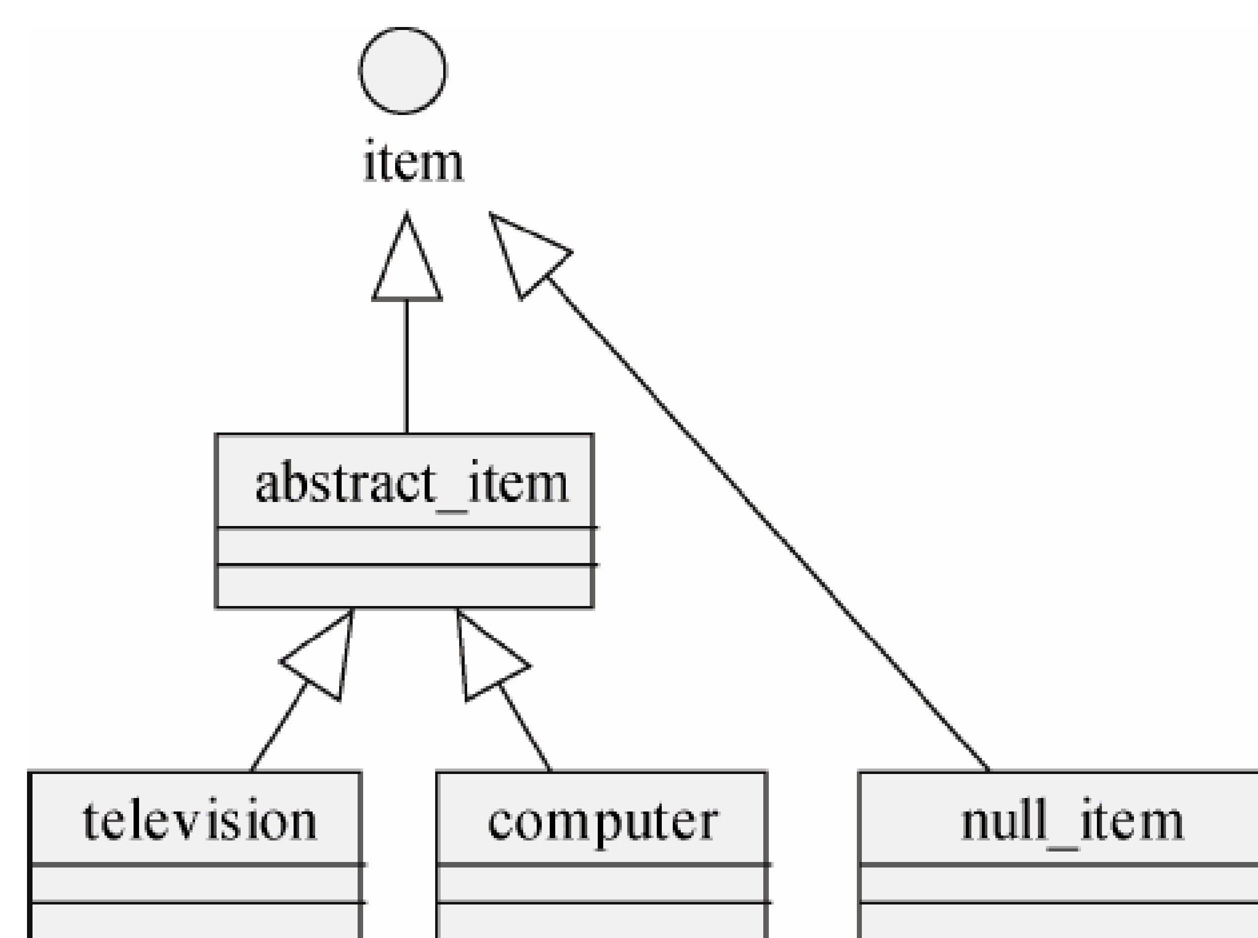


图 5-3　简单的多态类继承体系

```
class item : boost::noncopyable
{
```

```
public:
  virtual ~item(){}
  virtual void print() = 0;
};
class abstract_item : public item                    //抽象类
{
  string name;
public:
  abstract_item(const string& str):name(str){}
  virtual ~abstract_item(){}

  virtual void print()
  {   cout << name << endl; }
};

class television : public abstract_item
{
public:
  television():abstract_item("television")
  {}
};
class computer : public abstract_item
{
public:
  computer():abstract_item("computer")
  {}
};
class null_item: public item                          //空对象
{
  virtual void print(){}                              //什么也不做的空操作
};
```

因为基类使用了 boost::noncopyable，所有这些类都是不可拷贝的，无法被标准容器所容纳，但指针容器没有这些限制，所以可以管理它们。null_item 实现了 item 要求的所有接口，但没有任何有意义的操作，所以是一个空对象，可以当做一个空指针来使用。

```
int main()
{
  typedef ptr_vector<item> ptr_vec;      //指针容器，不使用 nullable<T>
  ptr_vec vec;
```

```
    vec.push_back(new television);          //添加对象
    vec.push_back(new computer);
    vec.push_back(new null_item);           //添加空对象，相当于空指针

    BOOST_FOREACH(item& i, vec)             //可以使用 BOOST_FOREACH 遍历容器
    {
        i.print();                          //无须空指针判断
    }

    vec.replace(2, new computer);           //空对象可以在适当的时候被替换成实对象
    vec[2].print();
}
```

程序的运行结果如下：

```
television
computer
computer
```

对比 5.9.3 小节的代码，很明显，空对象模式的使用简化了我们的操作代码。

读者还应该注意到这个例子中的类体系是 noncopyable 的（不能拷贝构造），并且没有定义 new_clone() 和 delete_clone() 函数，因此它们不支持可克隆概念，但仍然可以使用指针容器容纳。关于它们的进一步讨论可见 5.19.5 小节。

5.10　关联指针容器的共通能力

associative_ptr_container 是关联指针容器的基类，定义了一些关联指针容器的共有特征，它位于名字空间 boost::ptr_container_detail。

5.10.1　类摘要

associative_ptr_container 是 reversible_ptr_container 的子类，由于集合与映射这两种关联容器的差距较大，它只有很少的公开接口，大部分是保护的成员，类摘要如下（reversible_ptr_container 已有的未列出）：

```
template<class Config, class CloneAllocator>
class associative_ptr_container :
    public reversible_ptr_container<Config,CloneAllocator>
{
public:
    //类型定义
    typedef  Config::key_type key_type;
    typedef  Config::key_compare key_compare;
    typedef  Config::value_compare value_compare;
public:
    //函数对象访问
    key_compare key_comp() const;
    value_compare value_comp() const;
public:
    //删除操作
    iterator erase( iterator before );
    size_type  erase( const key_type& x );
    iterator erase( iterator first, iterator last );
};
```

5.10.2 接口解说

associative_ptr_container 有大量的内部保护成员供子类复用，仅提供了少量的公开接口，但这些接口都很重要。

associative_ptr_container 使用配置元函数增加了几个新的内部类型定义，它们是关联容器所必需的类型定义：

- key_type ：对于集合是值类型 value_type（即元素 T），对于映射是键的类型；
- key_compare ：键值比较函数对象的类型；
- value_compare：值的比较函数对象的类型，基本相当于 key_compare。

成员函数 key_comp()和 value_comp()分别用于返回比较函数对象的拷贝。

5.11 集合指针容器适配器

对于集合指针容器来说，它可以分为有序的和无序的，也可以分为不允许重复（单键）和允许重复的（多键），所以它的适配器比序列指针容器适配器要复杂一些。

ptr_container 库提供两个集合指针容器适配器，它们都位于头文件<boost/ptr_container/ptr_set_adapter.hpp>。

5.11.1 配置元函数

集合指针容器使用的配置元函数是 ptr_container_detail::set_config，它的类摘要如下：

```
template
<   class Key,                                          //集合元素类型
    class VoidPtrSet,                                   //被适配的集合容器类型
    bool  Ordered>                                      //是否有序
struct set_config
{
   typedef VoidPtrSet  void_container_type;             //内部集合容器容器
   typedef VoidPtrSet::allocator_type allocator_type;   //内存分配器

   typedef Key          value_type;                     //值类型
   typedef value_type  key_type;                        //键类型同值类型

   typedef mpl::eval_if_c<...>::type value_compare;     //有序专用
   typedef value_compare              key_compare;      //有序专用

   typedef mpl::eval_if_c<...>::type hasher;            //无序专用
   typedef mpl::eval_if_c<...>::type key_equal;         //无序专用

   typedef mpl::if_c<...>::type       container_type;

   typedef void_ptr_iterator<VoidPtrSet::iterator, Key > iterator;
   typedef void_ptr_iterator<VoidPtrSet::const_iterator, const Key >
               const_iterator;
};
```

set_config 有三个模板参数，Key 和 VoidPtrSet 与序列指针容器适配器的配置元函数 sequence_config（见 5.3.1 小节）含义相同，分别是容纳的元素类型和容纳 void* 指针的集合容器类型，bool 参数 Ordered 标志了集合是有序还是无序的。

set_config 接下来的元计算实现了集合指针容器所需的各种类型，这些元函数都非常简单，读者可参照 sequence_config。

还需要注意的是集合指针容器没有使用 nullable<T>系列元函数，因为对于集合这样的关联容器来说容纳空指针没有意义。我们不能使用 nullable<T>作为模板参数，也不能向集合指针容器插入空指针，这与序列指针容器相比是一个很大的不同。

5.11.2 ptr_set_adapter

ptr_set_adapter 用于适配不允许重复（单键）的集合容器，它的类摘要如下[①]：

```
template
<   class Key,
    class VoidPtrSet,
    class CloneAllocator = heap_clone_allocator,
    bool  Ordered        = true >
class ptr_set_adapter: public
      associative_ptr_container< set_config<Key,VoidPtrSet,Ordered>,
                                 CloneAllocator >
{
public:
    //集合特有的插入操作
    std::pair<iterator,bool>  insert( key_type* x );
    template< class U >
    std::pair<iterator,bool> insert( std::auto_ptr<U> x );

public:
    //指针所有权的转移
     template< class PtrSetAdapter >
    bool transfer( PtrSetAdapter::iterator object,
                   ptr_set_adapter& from );

     template< class PtrSetAdapter >
```

① 实际上 ptr_set_adapter 的真正父类是 ptr_set_adapter_base，这里做了适当的简化。

```
    size_type transfer( PtrSetAdapter::iterator first,
                        PtrSetAdapter::iterator last,
                       ptr_set_adapter& from );

    template< class PtrSetAdapter >
    size_type transfer( PtrSetAdapter& from );
};
```

ptr_set_adapter 是 reversible_ptr_container 的子类，具有 reversible_ptr_container 的全部能力，还增加了集合容器特有的插入操作。insert()在插入元素时会返回一个 std::pair，除了表明新位置的迭代器外还有一个 bool 值，用来表示插入操作是否成功，这是因为单键集合不允许重复的值，如果值已经存在则插入操作会失败。

ptr_set_adapter 同样提供一系列的 transfer()函数用于指针的所有权管理，它们的行为与序列指针容器完全相同，可参见 5.3.3 小节。

5.11.3 ptr_multiset_adapter

ptr_multiset_adapter 用于适配允许重复（多键）的集合容器，它的类摘要如下[①]：

```
template
<   class Key,
    class VoidPtrSet,
    class CloneAllocator = heap_clone_allocator,
    bool  Ordered        = true >
class ptr_multiset_adapter: public
      associative_ptr_container< set_config<Key,VoidPtrSet,Ordered>,
                                 CloneAllocator >
{
public:
    //多键集合特有的插入操作
    iterator  insert(key_type * x );
    template< class U >
    iterator  insert( std::auto_ptr<U> x );

public:
    //指针所有权的转移
```

① 实际上 ptr_multiset_adapter 的真正父类是 ptr_set_adapter_base，这里做了适当的简化。

```
    template< class PtrSetAdapter >
    bool transfer( PtrSetAdapter::iterator object,
                   ptr_set_adapter& from );

    template< class PtrSetAdapter >
    size_type transfer( PtrSetAdapter::iterator first,
                        PtrSetAdapter::iterator last,
                        ptr_set_adapter& from );

    template< class PtrSetAdapter >
    size_type transfer( PtrSetAdapter& from );
};
```

ptr_multiset_adapter 与 ptr_set_adapter 非常相似，它们的区别仅在于是否允许重复的值，因此 ptr_multiset_adapter 的插入操作可以直接返回迭代器，插入操作总会成功。

5.12 ptr_set 和 ptr_multiset

ptr_set 和 ptr_multiset 是有序集合指针容器，它们分别使用了标准容器 std::set 和 std::multiset 容纳 void*指针，位于头文件<boost/ptr_container/ptr_set.hpp>。

5.12.1 类摘要

ptr_set 和 ptr_multiset 的代码实现非常简单，因为大部分工作都已经在适配类中完成了，类摘要如下：

```
template
<   class Key,
    class Compare          = std::less<Key>,
    class CloneAllocator   = heap_clone_allocator,
    class Allocator        = std::allocator<void*> >
class ptr_set :
    public ptr_set_adapter< Key,
    std::set<void*,void_ptr_indirect_fun<Compare,Key>,Allocator>,
                        CloneAllocator, true >
```

```
{...};

template
<   class Key,
    class Compare          = std::less<Key>,
    class CloneAllocator   = heap_clone_allocator,
    class Allocator        = std::allocator<void*> >
class ptr_multiset :
    public ptr_multiset_adapter< Key,
 std::multiset<void*,void_ptr_indirect_fun<Compare,Key>,Allocator>,
                                 CloneAllocator, true >
{...};
```

代码中的 void_ptr_indirect_fun 是一个用于比较 void*指针的函数对象，它在执行比较操作时使用 static_cast<>把 void*转换为 Key 类型，然后再使用 Compare 进行比较，可参见 5.19.2 小节。

5.12.2 用法

ptr_set 和 ptr_multiset 用起来与标准容器 std::set 和 std::multiset 没有太大的差别，结合之前序列指针容器的使用经验可以很快掌握。再提醒一下，这些集合容器不能容纳空指针。

示范 ptr_set 和 ptr_multiset 用法的代码如下：

```
#include <boost/ptr_container/ptr_set.hpp>
using namespace boost;

int main()
{
  typedef ptr_set<string> ptr_set_t;              //有序集合指针容器
  ptr_set_t s;                                    //缺省构造
  assert(s.empty());                              //容器为空

  assert(s.insert(new string("fire")).second);    //插入元素
  assert(s.insert(new string("emblem")).second);
  assert(s.size() == 2);

  assert(!s.insert(new string("fire")).second);   //不允许重复元素
  assert(s.size() == 2);
```

```
    auto_ptr<ptr_set_t> ap = s.release();            //释放指针的所有权
    assert(s.empty());

    typedef ptr_multiset<string> ptr_mset_t;         //有序多键集合指针容器
    ptr_mset_t ms(ap->begin(), ap->end());           //克隆构造，应使用 transfer()
    assert(ms.size() == 2);

    ms.insert(new string("fire"));                   //允许多个重复元素
    ms.insert(new string("emblem"));
    assert(ms.size() == 4);
    assert(ms.count("fire") == 2);                   //使用 count()计算元素的个数
}
```

5.13 ptr_unordered_set 和 ptr_unordered_multiset

ptr_unordered_set 和 ptr_unordered_multiset 是无序集合指针容器，它们分别使用了 Boost 容器 boost::unordered_set 和 boost::unordered_multiset 容纳 void*指针，位于头文件<boost/ptr_container/ptr_unordered_set.hpp>。

5.13.1 类摘要

ptr_unordered_set 和 ptr_unordered_multiset 是散列容器，不保持元素的有序，因此它们的模板参数不同于标准容器，需要使用散列函数对象和相等函数对象，缺省是 boost::hash（见 4.1 小节）和 std::equal_to，类摘要如下：

```
template
<   class Key,
    class Hash              = boost::hash<Key>,
    class Pred              = std::equal_to<Key>,
    class CloneAllocator    = heap_clone_allocator,
    class Allocator         = std::allocator<void*> >
class ptr_unordered_set :
    public ptr_set_adapter< Key,
      boost::unordered_set<void*,void_ptr_indirect_fun<Hash,Key>,
                          void_ptr_indirect_fun<Pred,Key>,Allocator>,
                          CloneAllocator, false >
```

```
{
public:
    //散列容器函数对象类型定义
    typedef  Config::hasher hasher;
    typedef  Config::key_equal key_equal;

private:
    //有序容器相关的接口均禁用
    using base_type::rbegin;
    using base_type::rend;
    using base_type::crbegin;
    using base_type::crend;
    using base_type::key_comp;
    using base_type::value_comp;
    using base_type::front;
    using base_type::back;
public:
    using base_type::begin;
    using base_type::end;
    ...//其他集合操作函数
};

template
<   class Key,
    class Hash             = boost::hash<Key>,
    class Pred             = std::equal_to<Key>,
    class CloneAllocator   = heap_clone_allocator,
    class Allocator        = std::allocator<void*> >
class ptr_unordered_multiset :
      public ptr_multiset_adapter< Key,
  boost::unordered_multiset<void*,void_ptr_indirect_fun<Hash,Key>,
       void_ptr_indirect_fun<Pred,Key>,Allocator>,
                                CloneAllocator, false >
{...};    //代码同 ptr_unordered_set
```

ptr_unordered_set 和 ptr_unordered_multiset 通过声明父类接口为 private 的方式禁用了一些只有有序容器才能使用的接口，如 rbegin()、front()、back() 等。

5.13.2 用法

虽然内部实现机制不同，但 ptr_unordered_set 和 ptr_unordered_multiset 仍然具有集合容器的大部分标准接口，只是要注意集合是无序的这个特点，不能在无序容器上应用已序区间的算法。

示范 ptr_unordered_set 和 ptr_unordered_multiset 用法的代码如下，它们与 5.12.2 小节的很类似，只是改用了 transfer() 成员函数来转移指针的所有权：

```
#include <boost/ptr_container/ptr_unordered_set.hpp>
using namespace boost;

int main()
{
  typedef ptr_unordered_set<string> ptr_set_t;       //无序集合指针容器
  ptr_set_t s;
  assert(s.empty());

  assert(s.insert(new string("king")).second);       //插入元素
  assert(s.insert(new string("kong")).second);
  assert(s.size() == 2);

  assert(!s.insert(new string("king")).second);      //不允许重复元素
  assert(s.size() == 2);

  typedef ptr_unordered_multiset<string> ptr_mset_t;
  ptr_mset_t ms;

  ms.transfer(s);                        //从 s 中转移指针所有权到无序多键集合指针容器
  assert(ms.size() == 2);

  ms.insert(new string("king"));         //允许多个重复元素
  ms.insert(new string("kong"));
  assert(ms.size() == 4);
  assert(ms.count("king") == 2);
}
```

5.14 映射指针容器适配器

ptr_container 库的映射指针容器同样可以分为有序和无序、不允许重复（单键）和允许重复（多键），共有两个映射指针容器适配器，它们都位于头文件<boost/ptr_container/ptr_map_adapter.hpp>。

5.14.1 配置元函数

映射指针容器使用的配置元函数是 ptr_container_detail::map_config,它的类摘要如下：

```
template
<   class T,
    class VoidPtrMap,
    bool  Ordered >
struct map_config
{
    typedef remove_nullable<T>::type U;                     //容器的值类型
    typedef VoidPtrMap  void_container_type;                //被适配的容器类型

    typedef  VoidPtrMap::allocator_type allocator_type;

   typedef mpl::eval_if_c<...>::type  value_compare;        //有序专用
   typedef  mpl::eval_if_c<...>::type key_compare;          //有序专用

   typedef mpl::eval_if_c<...>::type  hasher;               //无序专用
   typedef mpl::eval_if_c<...>::type  key_equal;            //无序专用

   typedef mpl::if_c<...>::type       container_type;
    typedef  VoidPtrMap::key_type     key_type;             //键类型
    typedef U                         value_type;           //值类型

    typedef ptr_map_iterator<  VoidPtrMap::iterator,
                 key_type, U* const > iterator;
    typedef ptr_map_iterator<  VoidPtrMap::const_iterator,
                 key_type, const U* const> const_iterator;
};
```

由于都是关联容器，所以 map_config 与 set_config 相同，都有三个模板参数。T 和 VoidPtrMap 分别是映射容器的 value 类型（key 类型不在这里指定）和被适配的映射容器类型，bool 参数 Ordered 标志了映射是有序还是无序的。

map_config 同样元计算了映射指针容器所需的各种类型，只是迭代器类型不是之前一直使用的 void_ptr_iterator，这是因为映射容器容纳的元素是一个 std::pair。ptr_map_iterator 使用了 boost::iterator_adaptor（参见 3.5 小节）适配了标准映射容器的迭代器，返回一个模仿 std::pair 的 ref_pair 类型。

我们还需要注意类型 value_type，它使用了元函数 remove_nullable<T>，这意味着我们可以在映射容器中使用值的空指针。

5.14.2 ptr_map_adapter

ptr_map_adapter 用于适配不允许重复（单键）的映射容器，它的类摘要如下[①]：

```
template
<   class T,
    class VoidPtrMap,
    class CloneAllocator  = heap_clone_allocator,
    bool  Ordered         = true >
class ptr_map_adapter :
 public associative_ptr_container< map_config<T,VoidPtrMap,Ordered>,
                                   CloneAllocator >
{
public:
    //类型定义
    typedef  base_type::iterator         iterator;
    typedef  base_type::const_iterator   const_iterator;
    typedef  base_type::size_type        size_type;
    typedef  base_type::key_type         key_type;
    typedef  base_type::mapped_type      mapped_type;
    typedef  base_type::const_reference const_reference;
    typedef  base_type::auto_type        auto_type;
    typedef  VoidPtrMap::allocator_type allocator_type;
public:
    //插入操作
```

① 实际上 ptr_map_adapter 的真正父类是 ptr_map_adapter_base，这里做了适当的简化。

```
    std::pair<iterator,bool> insert( key_type& key, mapped_type x );

    template< class U >
    std::pair<iterator,bool> insert( const key_type& key, std::auto_ptr<U> x );

    iterator insert( iterator before, key_type& key, mapped_type x );

    template< class U >
    iterator insert( iterator before, const key_type& key, std::auto_ptr<U> x );

public:
    //访问元素
    mapped_reference at( const key_type& key );
    mapped_reference operator[]( const key_type& key );

public:
    //指针所有权的转移
    template< class PtrMapAdapter >
    bool transfer(  PtrMapAdapter::iterator object,
                PtrMapAdapter& from );

    template< class PtrMapAdapter >
    size_type transfer(  PtrMapAdapter::iterator first,
                    PtrMapAdapter::iterator last,
                   PtrMapAdapter& from );

    template< class PtrMapAdapter >
    size_type transfer( PtrMapAdapter& from );
};
```

因为映射容器存储的是 pair，所以 ptr_map_adapter 的类型定义较之前的容器有所不同。它的 value_type 是一个 ref_pair 类型，映射的左右类型分别定义为 key_type 和 mapped_type（即 T*），这与标准容器 std::map 是一致的。

ptr_map_adapter 不允许重复的键值，所以我们可以使用 at() 和 operator[] 来直接访问元素。注意：如果要使用 operator[] 来访问元素，那么要求容纳的元素类型 T 必须有一个缺省构造函数，而且不能是空指针，否则会发生编译错误[①]。

映射容器的成员函数 insert() 直接使用 mapped_type 的形式有点特别，它要求

① 这一点与使用标准映射容器容纳指针时有明显的不同，但与标准映射容器使用 operator[] 时的要求一致，读者需要特别留意。

key_type 必须是一个左值，这是出于异常安全的考虑，const 引用的形式必须使用 std::auto_ptr 来包装 new 的结果（示例代码参见 5.15.2 小节）。

指针转移操作与其他的指针容器都是类似的，不再赘述。

5.14.3 ptr_multimap_adapter

ptr_multimap_adapter 用于适配允许重复（多键）的映射容器，它的类摘要如下①：

```
template
<   class T,
    class VoidPtrMultiMap,
    class CloneAllocator = heap_clone_allocator,
    bool  Ordered        = true    >
class ptr_multimap_adapter :
 public associative_ptr_container< map_config<T,VoidPtrMap,Ordered>,
                                             CloneAllocator >
{
public:
    //类型定义
    typedef  base_type::iterator            iterator;
    typedef  base_type::const_iterator      const_iterator;
    typedef  base_type::size_type           size_type;
    typedef  base_type::key_type            key_type;
    typedef  base_type::mapped_type         mapped_type;
    typedef  base_type::const_reference     const_reference;
    typedef  base_type::auto_type           auto_type;
    typedef  VoidPtrMap::allocator_type     allocator_type;

public:
    //插入操作
    std::pair<iterator,bool> insert( key_type& key, mapped_type x );

    template< class U >
    std::pair<iterator,bool> insert( const key_type& key, std::auto_ptr<U> x );

    iterator insert( iterator before, key_type& key, mapped_type x );
```

① 实际上 ptr_multimap_adapter 的真正父类是 ptr_map_adapter_base，这里做了适当的简化。

```
    template< class U >
    iterator insert( iterator before, const key_type& key, std::auto_ptr<U> x );

public:
    //指针所有权的转移
    template< class PtrMapAdapter >
    bool transfer(  PtrMapAdapter::iterator object,
                   PtrMapAdapter& from );

    template< class PtrMapAdapter >
    size_type transfer(  PtrMapAdapter::iterator first,
                        PtrMapAdapter::iterator last,
                       PtrMapAdapter& from );

    template< class PtrMapAdapter >
    size_type transfer( PtrMapAdapter& from );
};
```

ptr_multimap_adapter 的类摘要几乎与 ptr_map_adapter 是相同的，只是不提供 at() 和 operator[] 操作，这是因为它允许重复的键值。

5.15 ptr_map 和 ptr_multimap

ptr_map 和 ptr_multimap 是有序映射指针容器，它们分别使用了标准容器 std::map 和 std::multimap 容纳 void* 指针，位于头文件 <boost/ptr_container/ptr_map.hpp>

5.15.1 类摘要

ptr_map 和 ptr_multimap 的代码实现非常简单，因为大部分工作都已经在适配类中完成了，类摘要如下：

```
template
<   class Key,
    class T,
    class Compare          = std::less<Key>,
    class CloneAllocator   = heap_clone_allocator,
```

```
    class Allocator       = std::allocator< std::pair<const Key,void*> >>
class ptr_map :
    public ptr_map_adapter<T,std::map<Key,void*,
                            Compare,Allocator>,CloneAllocator>
{...};

template
<   class Key,
    class T,
    class Compare          = std::less<Key>,
    class CloneAllocator   = heap_clone_allocator,
    class Allocator        = std::allocator< std::pair<const Key,void*> >>
class ptr_multimap :
    public ptr_multimap_adapter<T,std::multimap<Key,void*,
                                 Compare,Allocator>,CloneAllocator>
{...};
```

5.15.2 用法

ptr_map 和 ptr_multimap 用起来与标准映射容器 std::map 和 std::multimap 非常类似，大多数情况下我们只需要指定模板参数 Key 和 T 就可以正常工作了。

示范 ptr_map 和 ptr_multimap 用法的代码如下：

```
#include <boost/ptr_container/ptr_map.hpp>
using namespace boost;

int main()
{
  typedef ptr_map<int, string> ptr_map_t;          //有序映射指针容器
  ptr_map_t m;

  int a = 1;                                       //一个左值
  m.insert(a, new string("one"));                  //必须使用左值才能通过编译
  m.insert(10, auto_ptr<string>(new string("ten")));//自动指针可使用右值

  assert(m.size() == 2);
  assert(m[10] == "ten");                    //使用 operator[]，返回引用

  m.replace(m.begin(), new string("neo"));   //替换操作
```

```
    m[3] = "three";                        //使用 operator[]直接赋值,不用 new
    assert(m.at(1) == "neo" && m.at(3) == "three");

    BOOST_FOREACH(ptr_map_t::value_type& i, m) //foreach 循环
    {
        cout << *i->second << ",";             //second 是一个 T*指针
    }

    typedef ptr_multimap<int, string> ptr_multimap_t;//有序多键值映射
    ptr_multimap_t mm;

    mm.transfer(m.begin(), m);             //转移一个指针的所有权
    assert(m.size() == 2);                 //原容器失去所有权
    assert(mm.count(1) == 1);
}
```

如果允许存储空指针，那么需要注意 operator[]是无法使用的，可以改用 at()，例如：

```
typedef ptr_ map<int, nullable<string> > ptr_map_t;      //允许空指针
ptr_map_t m;

m.insert(10, auto_ptr<string>(new string("ten")));
m.at(10) = "tenten";                                     //编译正常
m[3] = "three";                                          //编译错误
```

5.16 ptr_unordered_map 和 ptr_unordered_multimap

ptr_unordered_map 和 ptr_unordered_multimap 是无序映射指针容器，它们分别使用了 Boost 容器 boost::unordered_set 和 boost::unordered_multiset 容纳 void*指针，位于头文件<boost/ptr_container/ptr_unordered_map.hpp>。

5.16.1 类摘要

ptr_unordered_map 和 ptr_unordered_multimap 是散列容器，需要使用散列函数对象和相等函数对象，与散列指针集合容器很相似，类摘要如下：

```
template
<   class Key,
```

```
    class T,
    class Hash              = boost::hash<Key>,
    class Pred              = std::equal_to<Key>,
    class CloneAllocator    = heap_clone_allocator,
    class Allocator         = std::allocator< std::pair<const Key,void*> >>
class ptr_unordered_map :
    public
ptr_map_adapter<T,boost::unordered_map<Key,void*,Hash,Pred,Allocator>,
                            CloneAllocator,false>
{
public:
    //散列容器函数对象类型定义
    typedef  Config::hasher hasher;
    typedef  Config::key_equal key_equal;

private:
    //有序容器相关的接口均禁用
    using base_type::rbegin;
    using base_type::rend;
    using base_type::crbegin;
    using base_type::crend;
    using base_type::key_comp;
    using base_type::value_comp;
    using base_type::front;
    using base_type::back;
public:
    using base_type::begin;
    using base_type::end;
    ...//其他集合操作函数
};
template
<   class Key,
    class T,
    class Hash              = boost::hash<Key>,
    class Pred              = std::equal_to<Key>,
    class CloneAllocator    = heap_clone_allocator,
    class Allocator         = std::allocator< std::pair<const Key,void*> >>
class ptr_unordered_multimap :
    public
ptr_multimap_adapter<T,boost::unordered_multimap<Key,void*,Hash,Pred,Alloca
```

```
tor>,
                                    CloneAllocator,false>
 {...};     //代码同 ptr_unordered_map
```

ptr_unordered_map 和 ptr_unordered_multimap 同样通过声明父类接口为 private 的方式禁用了一些只有有序容器才能使用的接口。

5.16.2 用法

如果读者熟悉了 ptr_map 和 ptr_unordered_set 的用法，那么 ptr_unordered_map 和 ptr_unordered_multimap 将会很容易掌握，因为它们既有映射容器的特性又有无序容器的特性。

示范 ptr_unordered_map 和 ptr_unordered_multimap 用法的代码如下：

```
#include <boost/ptr_container/ptr_unordered_map.hpp>
using namespace boost;

int main()
{
  typedef ptr_unordered_map<int, string> ptr_map_t;  //无序映射指针容器
  ptr_map_t m;

  int a = 1;
  m.insert(m.begin(),a, new string("one"));          //使用迭代器位置插入，左值
  m.insert(m.end(), 10,               //使用迭代器位置和 auto_ptr 插入，可用右值
              auto_ptr<string>(new stri ng("ten")));
  m[3] = "three";                                    //使用 operator[]

  assert(m.at(3) == "three");
  assert(m[10] == "ten");                            //使用 operator[]

  BOOST_FOREACH(ptr_map_t::value_type& i, m)         //同样可以使用 foreach
  {
      cout << *i->second << ",";
  }

  typedef ptr_unordered_multimap<int, string> ptr_multimap_t;
  ptr_multimap_t mm;
```

```
    mm.transfer(m.begin(), m.end(), m);          //指定迭代器区间转移所有权
    assert(mm.size() == 3);
    assert(m.empty() );
}
```

5.17 使用 assign 库

boost.assign 是一个可以快速方便地赋值或初始化容器的库，它使得向容器赋初值的操作变得异常简单，例如：

```
vector<int> v1, v2;                          //两个向量容器

using namespace boost::assign;               //打开 assign 名字空间
v1 += 1, 2, 3;                               //使用 operator+=和 operator,
push_back(v2)()(10), 20, 30;                 //使用 operator()和 operator,
```

ptr_container 库出现后 assign 库也提供了对指针容器赋值和初始化的支持，使我们可以不必再使用烦琐的 push_back()函数和 new 去填充元素。

5.17.1 向容器添加元素

assign 库在头文件<boost/assign/ptr_list_inserter.hpp>和<boost/assign/ptr_map_inserter.hpp>提供了四个向指针容器添加元素的辅助函数，分别为：

- ptr_push_back<T>() ：使用 push_back()在末端添加元素；
- ptr_push_front<T>()：使用 push_front()在前端添加元素；
- ptr_insert<T>() ：使用 insert()添加元素；
- ptr_map_insert<T>()：使用 insert()添加元素。

这些辅助函数接受一个指针容器的引用，返回一个 inserter 函数对象。inserter 重载了 operator()，支持最多五个参数（map 是六个，含键值），内部用操作符 new 和参数创建对象再调用指针容器的方法添加元素，如果没有参数则使用 T 的缺省构造函数来创建对象。所以我们无须再使用 new，同时还避免了映射容器对键值的左值要求。

通常情况下辅助函数创建的对象类型是容器的元素类型（即 T），是使用元编程技术自动

推导出来的，但如果有必要，辅助函数可以使用模板参数来明确指定创建的对象类型，这在指针容器容纳多态对象时是非常有用的，因为抽象类型无法实例化。

示范这些辅助函数用法的代码如下：

```
#include <boost/assign/ptr_list_inserter.hpp>
#include <boost/assign/ptr_map_inserter.hpp>
using namespace boost;

int main()
{
  using namespace boost::assign;                          //打开 assign 名字空间

  ptr_vector<int> v;
  ptr_push_back(v)()(1)(2)(100);                          //使用 ptr_push_back()
  assert(v.size() == 3);

  ptr_list<complex<double> > lt;
  ptr_push_front(lt)(1, 2)(0.618, 1.732);                 //使用 ptr_push_front()
  ptr_push_back<complex<double>>(lt)(2.718, 3.14);        //指明模板参数

  ptr_multimap<int, string> m;
  ptr_map_insert(m)(1, "one")(1, "neo");                  //使用 ptr_map_insert()
}
```

5.17.2 初始化容器元素

函数 ptr_list_of<T>()可以创建一个匿名的指针容器列表 generic_ptr_list<T>，在指针容器构造时直接初始化，较 ptr_push_back()的赋值方式效率更高，它位于头文件<boost/assign/ptr_list_of.hpp>。

ptr_list_of<T>()在使用时必须指定模板参数 T，也支持使用最多五个参数来创建对象，如果没有参数则使用 T 的缺省构造函数来创建对象，例如：

```
#include <boost/assign/ptr_list_of.hpp>
using namespace boost;

int main()
{
  using namespace boost::assign;                          //打开 assign 名字空间
```

```
  ptr_vector<int> v = ptr_list_of<int>()(1)(2);       //初始化向量指针容器

  ptr_deque<complex<double> > dq =                    //初始化双向队列指针容器
      ptr_list_of<complex<double>>(1, 2)(0.618, 1.732);

  ptr_set<string> s ;
  s = ptr_list_of<string>()("abc") ("xyz").to_container(s);//注意
}
```

generic_ptr_list<T>内部使用 ptr_vector 来存储元素，因此它可以很容易地搭配序列指针容器使用，对于集合指针容器则需要使用成员函数 to_container()进行转换（它也可以解决部分编译器的兼容问题），但对于映射指针容器就无能为力了。

令人遗憾的是 assign 库没有提供 ptr_map_list_of<T>()这样的函数，我们不能对一个映射指针容器直接初始化。

5.18 使用算法

ptr_container 库提供的指针容器都符合标准容器的要求，因此我们也可以在这些指针容器上应用算法，但因为指针容器存储元素的特殊性，不是所有的标准算法都适用，接下来将进行详细的讨论。

5.18.1 标准算法

C++标准库提供了近百个算法，可分为不变算法、修改算法、变序算法（含排序算法）、移除算法等类别，因为指针容器的迭代器屏蔽了内部的 void*指针，提供了间接访问接口，所以大部分算法都可以搭配指针容器工作。

对于一些比较重要的算法，指针容器提供了成员函数版本，它们通常能够比标准算法提供更好的运行效率，这些内部算法在 5.18.2 和 5.18.3 小节介绍。

不变算法

不变算法基本上都可以安全地应用于指针容器，下面的代码简单示范了 count、find、equal 等算法的使用，与标准容器无任何差异：

```
ptr_deque<int> dq;                                //双向队列指针容器
```

```
ptr_push_back(dq)(1)(2)(10)(10)(9);        //顺序插入元素

//计算元素的个数，输出 2
cout << std::count(dq.begin(), dq.end(), 10);

//计算满足条件的元素个数,使用了 bind,输出 3
cout << std::count_if(dq.begin(), dq.end(),
  boost::bind(std::greater<int>(), _1, 8));

//获取最小最大元素所在的位置，输出 1 和 10
cout << *std::min_element(dq.begin(), dq.end());
cout << *std::max_element(dq.begin(), dq.end());

//查找元素，输出 2
cout << *std::find(dq.begin(), dq.end(), 2);

//对容器内的元素求和，输出 32
cout << std::accumulate(dq.begin(), dq.end(), 0);

ptr_list<int> lt(dq.begin(), dq.end());  //克隆构造另一个指针容器

//比较两个容器是否相等
assert(std::equal(dq.begin(), dq.end(), lt.begin()));
```

for_each 算法也可以用于指针容器：

```
ptr_vector<item> vec;                      //使用 5.9.4 小节定义的类
...                                        //添加元素
std::for_each(vec.begin(), vec.end(),
  mem_fn(&item::print));                   //使用 mem_fn 调用成员函数，参见 4.2 节
```

修改算法

部分修改算法也可以应用于指针容器，但需要小心它们的变动语义，访问元素时要求元素必须是存在的而且可赋值，例如：

```
ptr_deque<int> dq;                                //双向队列指针容器
ptr_push_back(dq)(1)(2)(10)(10)(9);               //顺序插入元素

std::transform(dq.begin(), dq.end(), dq.begin(), //转换算法
  boost::bind(std::plus<int>(), _1, 3));          //把所有元素加 3
assert(dq.front() == 4);
```

```
std::replace(dq.begin(), dq.end(), 13, 20);       //替换算法
assert(dq[2] == 20);

std::fill(dq.begin(), dq.end(), 99);              //填充算法
assert(dq.back() == 99);

ptr_vector<int> vec;                              //向量指针容器
ptr_push_back(vec)(1)(2)(3);                      //添加 3 个元素

std::copy(dq.begin(), boost::next(dq.begin(), 3),//拷贝算法
      vec.begin());
assert(vec[1] == 99);
```

下面的代码会发生运行时异常：

```
ptr_vector<int> vec(10);                          //保留 10 个元素的空间
std::copy(dq.begin(), dq.end(), vec.begin());     //抛出异常
```

这是因为 ptr_vector 的构造函数不同于 std::vector 的构造函数，它仅仅是保留了空间，内部并没有真正分配保存元素，而 copy 算法使用的是覆盖语义，像未分配空间写入会发生错误。解决办法可以如前面代码那样预先添加元素，或者使用插入迭代器（参见 5.19.3 小节）：

```
std::copy(dq.begin(),dq.end(),
  ptr_container::ptr_back_inserter(vec));
```

变序算法和排序算法

变序算法和排序算法使用的是赋值和交换，因此要求元素必须是可赋值的，但因为指针容器中存储的是指针，因而效率没有内置的直接操作指针同名算法那么高效：

```
ptr_deque<int> dq;
ptr_push_back(dq)(20)(1)(2)(10)(9)(10)(100);      //顺序插入元素

//逆序算法，100,10,9,10,2,1,20
std::reverse(dq.begin(), dq.end());

//稳定排序，1,2,9,10,10,20,100
std::stable_sort(dq.begin(), dq.end());
```

```
//删除重复元素，搭配 erase，1,2,9,10,20,100
dq.erase(std::unique(dq.begin(), dq.end()),
  dq.end());

//删除元素，搭配 erase，1,2,9,10,100
dq.erase(std::remove(dq.begin(), dq.end(), 20),
  dq.end());

//随机打乱数据
std::random_shuffle(dq.begin(), dq.end());

//对前 3 个位置部分排序，1,2,9,100,10（后两个数字随机）
std::partial_sort(dq.begin(), boost::next(dq.begin(), 3), dq.end());
```

已序区间算法

只要不涉及元素的变动，已序区间算法就是不变算法，因而完全可以应用于指针容器：

```
ptr_deque<int> dq;
ptr_push_back(dq)(1)(2)(9)(10)(20);                 //添加已序元素

assert(std::binary_search(dq.begin(), dq.end(), 9)); //二分查找
cout << *std::lower_bound(dq.begin(), dq.end(), 3);  //下界
cout << *std::upper_bound(dq.begin(), dq.end(), 10); //上界

ptr_vector<int> vec;
ptr_push_back(vec) (2)(9)(10);                      //添加已序元素
assert(std::includes(dq.begin(), dq.end(),          //子集判定
  vec.begin(), vec.end()));
```

已序区间算法的 merge()、set_union()等算法因为涉及元素的变动，用于指针容器不够方便，需要搭配插入迭代器（参见 5.19.3 小节），通常使用成员函数会更好：

```
ptr_deque<int> dq;                                  //同前
ptr_vector<int> vec;                                //同前
ptr_list<int> lt;                                   //一个空的双向链表指针容器

std::set_union(dq.begin(), dq.end(),                //计算并集
  vec.begin(), vec.end(),
  ptr_container::ptr_back_inserter(lt));
```

```
lt.clear();

std::set_intersection(dq.begin(), dq.end(),//计算交集
  vec.begin(), vec.end(),
  ptr_container::ptr_back_inserter(lt));
```

5.18.2 序列指针容器的算法

ptr_sequence_adapter 为所有序列指针容器提供了下列四个内部算法：

- sort()　　：快速排序（使用 std::sort）；
- unique()　：删除相邻的重复元素；
- erase_if()：删除满足条件的元素（使用 std::remove_if）；
- merge()　：合并两个已序区间的元素（使用 std::inplace_merge）。

这些算法都有类似的形式，可以操作整个容器，也可以操作一个指定的区间，比较准则也可以自定义。

sort()

sort()算法对应于标准算法 std::sort()，用来对序列指针容器排序，它有四种形式，声明如下：

```
void sort();
void sort( iterator first, iterator last );
template< class Compare >
void sort( Compare comp );
template< class Compare >
void sort( iterator begin, iterator end, Compare comp );
```

sort()算法与标准排序算法 std::sort()很像，它可以对整个容器排序，也可以指定一个区间（但 ptr_list 不能指定区间）。示范 sort()算法的代码如下：

```
ptr_deque<int> dq;
ptr_push_back(dq)(100)(1)(2)(10)(9);        //添加元素

//对前 3 个元素排序，1,2,100,10,9
```

```
dq.sort(dq.begin(), boost::next(dq.begin(), 3));

ptr_list<int> lt(dq.begin(), dq.end());      //克隆构造

//使用大于比较准则排序，100,10,9,2,1
lt.sort(std::greater<int>());
```

unique()

unique()算法对应于标准算法 std::unique()，可以移除连续的重复元素，也有四种形式：

```
void unique();
void unique( iterator first, iterator last );
template< class Compare >
void unique( Compare comp );
template< class Compare >
void unique( iterator begin, iterator end, Compare comp );
```

unique()算法操作的不一定是已序区间，当然，如果排序后再执行 unique()则必定会删除所有重复的元素。

示范 unique()算法的代码如下：

```
ptr_vector<int> vec;
ptr_push_back(vec)(100)(1)(2)(2)(2)(10)(9)(7)(9);        //添加元素

//移除重复的元素，100,1,2,10,9,7,9
vec.unique();
```

erase_if()

erase_if()算法相当于标准算法 std::remove_if()搭配 erase()函数，可以删除某些特定的元素，它必须指定比较谓词。

erase_if()算法只有两种形式：

```
template< class Pred >
void erase_if( Pred pred );
template< class Pred >
void erase_if( iterator begin, iterator end, Pred pred );
```

示范 erase_if()算法的代码如下：

```
ptr_vector<int> vec;
ptr_push_back<int>(vec)(100)(1)(2)(2)(2)(10)(9)(7)(9);   //添加元素

//删除所有等于 2 的元素，100,1,10,9,7,9
vec.erase_if(boost::bind(std::equal_to<int>(), _1, 2));

//删除前 4 个中大于 1 的元素，1,7,9
vec.erase_if(vec.begin(), boost::next(vec.begin(), 4),
        boost::bind(std::greater<int>(), _1, 1));
```

merge()

merge()是一个已序区间算法，要求参与合并的两个容器都已经使用 BinPred 谓词排序，最后得到一个已序的容器，容器 from 中的元素被转移到目标容器。

merge()算法的声明如下：

```
void merge( ptr_sequence_adapter& r );
template< class BinPred >
void merge( ptr_sequence_adapter& r, BinPred comp );
void merge( iterator first, iterator last, ptr_sequence_adapter& from );
template< class BinPred >
void merge( iterator first, iterator last, ptr_sequence_adapter& from,
       BinPred comp );
```

示范 merge()算法的代码如下：

```
ptr_vector<int> vec;
ptr_push_back<int>(vec)(1)(2)(7)(9)(10);

ptr_vector<int> v2;
ptr_push_back(v2)(3)(5)(100);

//合并两个容器，1,2,3,5,7,9,10,100
v2.merge(vec);
assert(vec.empty());                  //vec 的指针被转移
```

5.18.3　关联指针容器的算法

关联指针容器分为有序和无序两种，因而支持的算法也不相同。

所有关联指针容器都提供如下三个算法：

- find(k)　　　　：查找键值 k，返回迭代器；
- count(k)　　　 ：计算键值 k 的数量；
- equal_range(k)：返回一个迭代器区间，里面的所有元素键值都是 k。

有序关联指针容器额外提供下面两个算法：

- lower_bound(k)：大于等于 k 的元素“下界”，返回第一个满足>=k 的迭代器；
- upper_bound(k)：大于 k 的元素“上界”，返回第一个满足>k 的迭代器。

对于有序关联指针容器来说，equal_range(k) 相当于 (lower_bound(k), upper_bound(k))。

find()和 count()

find()和 count()算法相当简单，它们相当于标准映射容器提供的同名函数，声明如下：

```
iterator       find( const key_type& x );
const_iterator find( const key_type& x ) const;
size_type      count( const key_type& x ) const;
```

find()和 count()算法用起来也很容易，例如：

```
ptr_unordered_set<int> us;          //无序单键集合容器
ptr_insert(us)(3)(5)(7)(13);

assert(us.find(5) != us.end());     //查找元素
assert(us.find(10) == us.end());
assert(us.count(10) == 0);          //计算数量

ptr_multimap<int, int> mm;          //有序多键映射容器
ptr_map_insert(mm)(1, 1)(1, 5)(2, 2)(3, 4);
```

```
assert(mm.count(1) == 2);          //计算数量
assert(*mm.find(3)->second == 4);  //查找元素
```

equal_range()

equal_range()返回一个迭代器区间，主要用于允许重复的关联指针容器，可以一次性得到所有键值等于 k 的元素，相当于 find()算法的泛化。

equal_range()算法的声明如下：

```
iterator_range<iterator>       equal_range( const key_type& x );
iterator_range<const_iterator> equal_range( const key_type& x ) const;
```

equal_range()返回一个 iterator_range 对象，它是一个具有类似容器接口的迭代器区间，比简单的 std::pair<iterator>功能强很多。

示范 equal_range()算法的代码如下：

```
ptr_multiset<int> us;                       //有序多键集合容器
ptr_insert(us)(3)(5)(7)(13)(3);

assert(us.count(3) ==2);

BOOST_AUTO(r1, us.equal_range(3));          //获得迭代器区间
assert(!r1.empty() );                       //可以判断区间是否空
assert(r1.front() == 3);                    //具有类似容器的接口

ptr_unordered_multimap<int, int> mm;        //无序多键映射容器
ptr_map_insert(mm)(1, 1)(1, 5)(2, 2)(3, 4);

BOOST_AUTO(r2, mm.equal_range(1));          //获得迭代器区间
for (BOOST_AUTO(p, r2.begin()); p != r2.end(); ++p)
{
  cout << *p->second << ",";                //输出 1, 5
}
```

lower_bound()和 upper_bound()

lower_bound()和 upper_bound()算法只能用于有序映射容器，它们的声明如下：

```
iterator lower_bound( const key_type& x );
```

```
const_iterator  lower_bound( const key_type& x ) const;
iterator        upper_bound( const key_type& x ) ;
const_iterator  upper_bound( const key_type& x ) const ;
```

它们的用法比较简单，下面的代码示范了它们的用法：

```
ptr_set<int> us;
ptr_insert(us)(3)(5)(13)(7)(2);          //集合容器会自动排序

assert(*us.lower_bound(4) == 5);
assert(*us.upper_bound(4) == 5);
assert(*us.upper_bound(5) == 7);
```

5.19 其他议题

指针容器是一个比较庞大的库，本节我们将讨论关于指针容器的一些其他议题，但并未囊括所有相关的内容。

5.19.1 异常

ptr_container 库是强异常安全的，如果出现访问空指针或者不存在的键值等情况会以抛出异常的方式通知用户。

ptr_container 库提供了三个异常类，它们都是 std::exception 的子类：

- bad_ptr_container_operation：最通用的指针容器异常，表示发生的操作错误；
- bad_index ：序列指针容器使用整数索引时出错；
- bad_pointer ：不允许使用空指针时使用空指针出错。

ptr_container 库内部使用宏 BOOST_PTR_CONTAINER_THROW_EXCEPTION 来抛出异常，缺省情况下它直接使用 throw 关键字抛出异常，但如果我们定义了宏 BOOST_NO_EXCEPTIONS 或者 BOOST_PTR_CONTAINER_NO_EXCEPTIONS，那么它将转化为断言 BOOST_ASSERT 从而完全禁用异常。

禁用异常适用于不允许使用异常的情形，同时也可能带来少量的性能提升，但除非有必

要不建议禁用异常，因为异常已经成为了 C++的一部分，禁用它不会带来太多的好处。

5.19.2 间接函数对象

为了更方便地操作指针或智能指针所指的对象，ptr_container 库在头文件 <boost/ptr_container/indirect_fun.hpp> 提供了两个函数对象适配器 indirect_fun 和 void_ptr_indirect_fun，它们可以把一个单参或者双参函数对象适配成可以直接操作指针的函数对象。

这两个函数对象的声明如下：

```
template< class Fun >
struct indirect_fun                                        //函数对象
{
    indirect_fun();
    indirect_fun( Fun f );

    template< class T >
    typename result_of< Fun( typename pointee<T>::type ) >::type
    operator()( const T& r ) const;

    template< class T, class U >
    typename result_of< Fun( typename pointee<T>::type,
                            typename pointee<U>::type ) >::type
    operator()( const T& r, const U& r2 ) const;
};

template< class Fun >
inline indirect_fun<Fun> make_indirect_fun( Fun f )       //工厂函数
{
    return indirect_fun<Fun>( f );
}

template< class Fun, class Arg1, class Arg2 = Arg1 >
struct void_ptr_indirect_fun                               //函数对象
{
    void_ptr_indirect_fun();
    void_ptr_indirect_fun( Fun f );
```

```
    typename result_of< Fun( Arg1 ) >::type
    operator()( const void* r ) const;

    typename result_of< Fun( Arg1, Arg2 ) >::type
    operator()( const void* l, const void* r ) const;
};

template< class Fun, class Arg >
inline void_ptr_indirect_fun<Fun,Arg>
make_void_ptr_indirect_fun( Fun f )                //工厂函数
{
    return void_ptr_indirect_fun<Fun,Arg>( f );
}
```

使用这两个函数对象，我们可以很容易地操作指针所指的对象，例如：

```
#include <boost/smart_ptr.hpp>
#include <boost/ptr_container/indirect_fun.hpp>
using namespace boost;

int main()
{
  typedef shared_ptr<string> ptr_string;          //智能指针

  ptr_string s1(new string("chrono"));
  ptr_string s2(new string("trigger"));

  assert(indirect_fun<not_equal_to<string> >()(s1, s2));
  cout << indirect_fun<std::plus<string> >()(s1, s2) << endl;

  void* p1 = s1.get();                             //void*指针
  void* p2 = s2.get();

  assert((void_ptr_indirect_fun<less<string>, string >()(p1, p2)));
}
```

5.19.3　插入迭代器

C++标准库为标准容器提供了三种插入迭代器，可以把容器适配成迭代器应用于算法，例如：

```
vector<int> v1, v2;
...                                         //某些赋值操作
std::copy(v1.begin(), v1.end(),             //copy 算法
  std::back_inserter(v2));                  //使用末端插入迭代器
```

ptr_container 库在头文件<boost/ptr_container/indirect_fun.hpp>提供了等价的三种指针插入迭代器和辅助函数，它们位于名字空间 boost::ptr_container：

- ptr_back_inserter(cont) ：使用 push_back()克隆指针然后插入；
- ptr_front_inserter(cont) ：使用 push_front()克隆指针然后插入；
- ptr_inserter(cont, before)：使用 insert()克隆指针然后插入。

这三种指针插入迭代器的用法与标准库的插入迭代器用法完全相同。

```
#include <boost/ptr_container/ptr_inserter.hpp>
using namespace boost;

int main()
{
  ptr_vector<int> v1, v2;

  using namespace boost::assign;            //打开 assign 名字空间
  v1 = ptr_list_of<int>()(1)(2)(100);       //赋值

  std::copy(v1.begin(), v1.end(),
      boost::ptr_container::ptr_back_inserter(v2));
}
```

5.19.4 使用视图分配器

之前我们介绍的所有指针容器都使用的是 heap_clone_allocator（参见 5.1.4 小节），它是 ptr_container 库缺省使用的容器，本小节我们将简单了解一下 view_clone_allocator 的用法。

view_clone_allocator 并不真正管理内存，而是提供一个只读的“指针视图”，因此我们可以使用它创建一个“视图容器”，它可以安全地用另一个容器的视角去观察原容器，不会造成任何影响。

示范 view_clone_allocator 用法的代码如下：

```
int main()
{
  using namespace boost::assign;                          //打开 assign 名字空间
  ptr_vector<int> v = ptr_list_of<int>(100)(1)(10) (2);  //向量指针容器

  ptr_set<int,std::less<int>, boost::view_clone_allocator>
      view(v.begin(), v.end());                         //使用有序集合指针容器视图

  BOOST_FOREACH(int& i, view)                           //foreach 遍历
  {
      cout << i << ",";
  }
}
```

这段代码中先创建了一个向量指针容器，然后用 view_clone_allocator 在其上建立了一个有序集合指针容器视图，这样就可以以有序集合的方式来只读地访问向量中的元素。

代码的运行结果如下：

```
1,2,10,100,                                   //有序集合重新整理了元素的顺序
```

5.19.5 可克隆性的再讨论

可克隆性（cloneable）是 ptr_container 库的一个很重要的概念，相当于标准容器对元素可拷贝的要求，但它不是指针容器所必须的，因为指针容器是泛型的，如果对元素的操作不涉及克隆（克隆构造、克隆赋值等）就不会使用克隆分配器，也就不涉及可克隆概念。

5.9.4 小节已经示范了不支持克隆概念类用于指针容器的情形，下面我们再来看一些代码：

```
int main()
{
  typedef ptr_vector<item> ptr_vec;         //使用之前的 item 类定义
  ptr_vec vec;

  using namespace boost::assign;
  ptr_push_back<television>(vec)();
  ptr_push_back<computer>(vec)();
```

```
    ptr_vec v2;
    v2.transfer(v2.begin(), vec.begin(), vec);      //可以使用转移所有权操作
}
```

但下面的代码由于使用了克隆操作所以无法通过编译：

```
ptr_vec v2(vec);                                    //克隆构造
v2 = vec.clone();                                   //调用克隆方法
```

总的来说，由于指针容器仅存储指针，所以大大降低了对元素的要求，虽然有的操作需要使用克隆，但因为存在 new_clone()和 delete_clone()函数的缺省实现，几乎无须任何多余的工作就可以使用 ptr_container 库的全部能力。不过对于不可拷贝的类型来说，必须像在标准容器里使用它们时一样谨慎。

5.19.6 序列化

ptr_container 库的所有指针容器都支持序列化，容器的序列化头文件位于目录<boost/ptr_container/>下，详细的讨论见第 9 章 9.8.5 小节。

5.20 总结

本章讨论了 Boost 库中的指针容器，它专门用于容纳指针元素，提供了与标准容器等价的 ptr_vector、ptr_list 等各种容器，而且是异常安全的，保证没有内存泄漏。

ptr_container 库里的指针容器基本与标准容器对应，接口和用法也非常相近，但它的根本特点是容器里容纳的是指针而不是元素的拷贝，因此存在许多与标准容器不同的地方。

指针容器对元素的要求非常低，不需要元素是可拷贝或可赋值的，因此几乎可以容纳任意类型的元素——包括 auto_ptr。指针容器也可以存储多态对象的指针，成为一个多态容器，这在很多时候都是非常有用的。

谈到指针就离不开空指针，虽然指针容器允许使用包装类 nullable<T>来容纳空指针，但这种做法很不安全，也很容易出错，应该尽量使用空对象模式来代替空指针的使用。

克隆是指针容器的一个特有概念，它类似标准容器中的内存分配器概念（allocator），可以在需要的时候创建指针所指对象的拷贝。ptr_container 库提供缺省的克隆分配器实现，但这不是必须的，很多指针容器的操作无须克隆概念也可以工作。

ptr_container 库提供了数量众多的指针容器，因此必须对指针容器的优缺点做到心中有数，充分发挥它们的优点同时避免它们的缺点。在使用时首先要决策是使用标准容器还是指针容器，标准容器容纳智能指针用法通用灵活，而指针容器有指针不可共享的限制，但用起来更方便。第二个决策是使用何种指针容器，由于指针容器与标准容器基本等价，因此可以参照标准容器的使用策略，最常用的是向量指针容器 ptr_vector。

ptr_container 库内容十分丰富，可以说与 STL 不相上下，读者可在阅读完本章后进一步参考 Boost 文档和源码深入研究。

第 6 章

侵入式容器

本章我们将看到另外一类新式容器：侵入式容器。

侵入式容器名字中的“侵入”一词容易令人产生不愉快的联想，但它并没有任何恶意。侵入式容器也是一类用于容纳元素的容器，但元素必须做出一些代码上的适度修改才能被容纳，因此，侵入式容器看起来像是“侵犯/闯入”了元素的内部实现，而实际上，我们对侵入式容器并不陌生，最早在数据结构课程中实现的链表、二叉树等数据结构都属于侵入式容器的范畴。

与侵入式容器相对应的是非侵入式容器。标准容器和第 5 章介绍的指针容器都属于非侵入式容器，这类容器不要求对容纳的元素做任何修改即可容纳，较侵入式容器实现手法要温和很多。因为非侵入式容器用起来简单方便，自 C++标准库出现后非侵入式容器逐渐大行其道，侵入式容器却日薄西山。

但侵入式容器也有自己的优点，它没有非侵入式容器的拷贝、克隆等要求，也可以保存抽象类，对内存管理的要求很低，而且允许定制数据结构，所以通常可以提供更好的性能。Boost 使用库 intrusive 重新引入了侵入式容器，而且具有近似标准容器的接口，大大降低了应用的难度，值得我们去学习使用。

6.1 概述

在这一节中我们将展示 intrusive 库的大致轮廓，了解它的一些基本概念。为了让读者对侵入式容器有一个比较明晰的印象，我们先来回顾一下数据结构的知识。

6.1.1 手工实现链表

读者应该对数据结构不会陌生，许多学校都会教授这个课程，线性表、链表、二叉树、堆等数据结构是计算机科学的基础，构建这些数据结构是一个程序员应该具备的基本素质。遗憾的是自从完整精致的 STL 横空出世，这一基本功就渐渐被大多数人荒废了（笔者也是其中之一，惭愧）。下面我们来实现一个最简单的单向链表结构，它是一个最简单的侵入式容器。

首先定义一个简单的节点类 point：

```
class point: boost::noncopyable          //简单的节点类，不可拷贝和赋值
{
public:
  int x,y;                               //节点的“有效载荷”

  //下面的代码是构造链表所需的“附件”
  typedef point* node_ptr;               //指针类型定义
  node_ptr next;                         //后继指针

  point(int a = 0, int b = 0):           //构造函数
      x(a), y(b), next(NULL){}           //后继初始化为空指针，即无后继

  node_ptr get_next()                    //获得后继节点
  {       return next;}

  void set_next(node_ptr p)              //设置后继节点
  {       next = p;}
};
```

point 类的真正作用是保存坐标值，但为了实现链表必须增加一个额外的存储空间：后继指针变量 next。为了规范指针的使用，我们定义了两个 get/set 成员函数，这种间接层比直接操作指针要好的多。[①]

下面的代码示范了节点类是如何连接成链表并使用的：

```
int main()
{
  point p1, p2(2,2), p3(3,3);            //三个节点对象，未连接

  p1.set_next(&p2);                      //p1 连接 p2，链表形如 p1->p2
  p2.set_next(&p3);                      //p2 连接 p3，链表形如 p1->p2->p3
```

① 后面我们会看到 intrusive 库同样使用了这样的实现手法。

```
    for (point::node_ptr p = &p1;         //指向头节点
         p != NULL;                       //循环终止条件
         p = p->get_next())               //指针前进到下一个节点
    {
        cout << p->x << "-" << p->y << " ";
    }

    p1.set_next(&p3);                     //从链表中移除 p2，链表形如 p1->p3
}
```

这个简单的例子展示了侵入式容器的一些重要特性：

首先，元素除了它自身的必备功能外还必须要增加一些额外的能力（这里是指向后继节点的指针）才能被纳入侵入式容器（链表）。其次，侵入式容器不负责内存分配，元素的创建是容器之外的事情，与它无关。第三，侵入式容器并不真正的“容纳”对象，元素仍然散落在内存中的各个位置，仅仅是使用指针以某种算法（这里是单向顺序算法）把它们连接起来便于访问而已，某种程度上把它称为“链接视图”或许会更恰当。最后，侵入式容器的插入删除等操作也只是操纵链接的指针，调整元素的链接顺序，不涉及内存的分配管理。

6.1.2 intrusive 库介绍

intrusive 库位于名字空间 boost::intrusive，由数个不同的头文件组成，实现了许多非常有用的侵入式容器，它们都位于目录<boost/intrusive/>下，使用具体的容器时包含所需头文件即可。

因为侵入式容器都是使用指针链接实现的，所以也可以称为“链式容器”——不存在与 std::vector、std::deque 等价的基于数组实现的容器。

intrusive 库提供的侵入式容器都具有与标准容器类似的接口，包括：

- slist : 单向链表；
- List : 类似 std::list 的双向链表，是最常用的侵入式容器；
- set/multiset/rbtree : 基于红黑树的类似 std::set 的关联容器；
- avl_set/avl_multiset/avltree：基于 AVL 树的类似 std::set 的关联容器；
- splay_set/splay_multiset/splaytree：基于 splay 树的类似 std::set 的关联容器；

- sg_set/sg_multiset/sgtree ：基于 scapegoat 树的类似 std::set 的关联容器；
- treap_set/treap_multiset/treap：基于堆二叉树的类似 std::set 的关联容器；
- unordered_set/unordered_multiset：无序关联容器。

根据侵入的程度这些容器又可分为纯侵入式容器（intrusive container）和半侵入式容器（semi-intrusive container）。

intrusive 库提供的大多数容器都属于纯侵入式容器（intrusive container），仅仅调整节点中的链接指针，并没有“容纳”任何东西。unordered_set 和 unordered_multiset 这两个无序关联容器属于半侵入式容器，这是因为它们需要一个额外的内存空间来维护散列容器所需的负载因子（load factor），不是完全的侵入。

侵入式容器的一个重要特点是它不负责管理元素的生命周期，仅仅是调整指针的链接，因此没有对象拷贝的运行开销，元素的创建工作被外部化，这与标准容器（使用内存分配器）和指针容器（使用克隆分配器）是明显不同的。带来的好处是最小化了内存使用，提高了运行效率。但这也同时是侵入式容器的缺点，因为内存管理的工作终归是有人要做，那么现在就交给了用户——元素的创建与删除对于侵入式容器来说是一个重要的工作，有些时候我们可以使用第 5 章的指针容器来简化。更进一步的推论是，侵入式容器与容纳的元素两者的生存期是不一致的，有可能因为元素被销毁而导致访问失败，所以元素的生命周期应该比侵入式容器要长。

6.2 入门示例

使用侵入式容器必须要修改自有类的定义，为此 intrusive 库提供了挂钩（hook）这一方便的工具类，它包含了一些必要的数据，可以把它近似地理解为链接指针的聚合。挂钩可以分为基类挂钩（base hook）和成员挂钩（member hook）两类，可以以基类或者成员的方式嵌入到自有类中使用。

下面我们使用单链表来示范侵入式容器的基本用法，其中涉及的概念参见 6.3 小节。

6.2.1 使用基类挂钩

单链表侵入式容器使用的基类挂钩是 slist_base_hook，并不需要我们做太多的工作，只要从它继承就可以了。point 类可以改写成如下的形式：

```
#include <boost/intrusive/slist.hpp>          //单链表侵入式容器
using namespace boost::intrusive;              //侵入式容器的名字空间

class point: public slist_base_hook<>          //使用基类挂钩，缺省配置
{
public:
  int x,y;                                     //无须自定义链接指针，已由挂钩实现
  point(int a = 0, int b = 0):
      x(a), y(b){}
};
```

单链表侵入式容器 slist 具有类似标准容器的接口，可以这样使用：

```
int main()
{
  point p1, p2(2,2), p3(3,3);     //3 个节点对象，未连接

  slist<point> sl;                //声明一个单链表侵入式容器，缺省配置，可容纳 point

  sl.push_front(p1);              //缺省情况下不能使用 push_back()
  sl.push_front(p2);
  sl.push_front(p3);              //链表形如 p3->p2->p1

  assert(sl.size() == 3);
  sl.reverse();                   //提供内置的逆序算法，链表形如 p1->p2->p3

  BOOST_FOREACH(point& p, sl)     //可以使用 foreach 算法
  {
      cout << p.x << "-" << p.y << " ";
  }

  sl.erase(boost::next(sl.begin()));//删除链表中的第二个节点
}
```

slist 是 intrusive 库提供的一个单链表容器，缺省情况下只能使用 push_front() 添加元素，如果在 slist 的模板参数中增加一个配置选项 cache_last<true>，那么它也可以使用 push_back()：

```
  slist<point, cache_last<true> > sl;     //使用元数据选项配置
  sl.push_back(p1);                       //可以使用 push_back()
```

6.2.2 使用成员挂钩

基类挂钩是使用侵入式容器最简单的方式，但有的时候我们可能不想使用基类挂钩，因为这种继承关系可能不是我们想要的。这时可以使用成员挂钩，把挂钩作为自有类的一个

public 成员：

```
class point                                          //不使用基类挂钩
{
public:
  ...                                                //同前
  slist_member_hook<> m_hook;                        //成员挂钩，缺省配置
};
```

侵入式容器在容纳使用成员挂钩的类时要多做一些工作，需要用一个 member_hook<> 配置选项，告诉容器成员挂钩的类型和挂钩变量名[①]：

```
  slist<point,                                       //元素类型
      member_hook<point,                             //成员挂钩选项
          slist_member_hook<>,&point::m_hook>,       //成员挂钩变量
      > sl;
```

使用成员挂钩的完整代码如下，这里我们还使用了指针容器来动态创建对象：

```
#include <boost/ptr_container/ptr_vector.hpp>
#include <boost/assign/ptr_list_of.hpp>
#include <boost/intrusive/slist.hpp>
using namespace boost::intrusive;
using namespace boost;

int main()
{
  using namespace boost::assign;                     //assign 名字空间
  ptr_vector<point> vec =                            //指针容器
      ptr_list_of<point>()(2,2)(3,3);                //使用 assign 库初始化

  typedef member_hook<point, slist_member_hook<>,    //使用成员挂钩 typedef
      &point::m_hook> member_option;                 //简化类型定义
  slist<point, member_option> sl;                    //单链表侵入式容器

  BOOST_REVERSE_FOREACH(point& p, vec)               //逆序遍历指针容器
  {
      sl.push_front(p);                              //使用 push_front()
  }
  assert(sl.size() == 3);
```

① 实际上这相当于我们手工处理了成员变量指针类型，分解出类类型和成员变量类型，因为编译器不具备从成员变量指针中推导出这些类型的能力，算是个“无奈之举”吧。

```
    BOOST_FOREACH(point& p, sl)          //遍历侵入式容器
    {
        cout << p.x << "-" << p.y << " ";
    }
}
```

这段代码中我们需要注意侵入式容器与元素的生命周期问题，指针容器及里面的元素必须先于侵入式容器创建，如果把两者的声明顺序调换一下，像这样：

```
slist<...> sl;                       //先声明侵入式容器
ptr_vector<point> vec = ...;         //后声明指针容器并插入元素
```

那么在 main() 函数结束时指针容器 vec 和里面的元素会先被销毁，当侵入式容器 slist 销毁时会因为访问不存在的元素而抛出异常。

如果要避免这个错误，可以在程序结束前调用侵入式容器的成员函数 clear()，提前“清空”容器。实际上它并不删除元素，仅是把元素间的链接关系断开而已：

```
sl.clear();                          //程序结束前调用，此时元素还是有效的
```

或者我们再改变一下挂钩和侵入式容器的配置选项，允许元素在析构时自动从容器中断开连接：

```
class point
{
public:
    //使用自动断开连接的选项
    slist_member_hook<link_mode<auto_unlink>> m_hook;
};

typedef member_hook<point,
        slist_member_hook<link_mode<auto_unlink> >,          //挂钩类型
        &point::m_hook> member_option;

//自动断开连接功能需禁用常量时间的 size() 功能
slist<point, member_option, constant_time_size<false> > sl;
```

6.3 基本概念

intrusive 库抽象了侵入式容器的实现手法，使用了大量的元编程技术，本节将介绍 intrusive 库中使用的这些基本概念。

6.3.1 节点

节点（node）概念抽象了手工实现链表时的链接指针和其他信息，把它们封装为一个类：单链表提供一个指针（后继），双向链表提供两个指针（前驱和后继），树结构则提供三个指针（父指针和左右指针）和颜色、平衡等其他信息。

有了节点概念，用户就可以直接以继承或者成员的方式复用它而无须自行编写侵入代码。例如，单链表 slist 的节点代码如下：

```
template<class VoidPointer>
struct slist_node
{
   typedef typename boost::pointer_to_other
      <VoidPointer, slist_node>::type node_ptr;   //节点指针类型
   node_ptr next_;                                //节点指针
};
```

因为使用了元函数 pointer_to_other<>（参见 2.5.3 小节）来定义节点指针类型，所以节点不仅可以使用原始指针，也能够使用智能指针。[1]

6.3.2 节点特征

节点特征（node traits）是一个 traits 类，它封装了操作节点的基本方法，对外提供了一致的抽象接口。不同的节点特征有不同的接口，但通常都提供节点类型、节点指针类型、取节点的前驱后继（静态成员函数）等操作。

例如，单链表 slist 的节点特征代码如下：

```
template<class VoidPointer>
struct slist_node_traits
{
   typedef some_define node;                              //节点类型
   typedef some_define node_ptr;                          //节点指针类型
   typedef some_define const_node_ptr;                    //节点常指针类型

   static node_ptr get_next(const_node_ptr n);            //取后继节点
   static void set_next(node_ptr n, node_ptr next);       //设置后继节点
};
```

① 不是所有智能指针都能被用于侵入式容器，特别是共享语义的智能指针是不可以的（如 shared_ptr）。

读者可以对比一下 6.1.1 小节的代码比较两者之间的异同。

6.3.3 节点算法

节点算法（node algorithms）抽象了链式数据结构的算法，使用节点特征类来操作节点，以静态成员函数的形式提供链表的一些基本操作，如初始化、连接节点、断开连接节点等等，用来实现某个特定数据结构——从简单的单链表到复杂的红黑树、AVL 树。节点算法与配套的节点特征类必须是接口兼容的（静态多态），否则就会在编译时发生错误。

例如，单链表 slist 的循环节点算法 circular_slist_algorithms 的部分代码如下：

```
template<class NodeTraits>                                      //使用节点特征类
class circular_slist_algorithms
{
public:
    typedef NodeTraits::node                 node;              //节点类型
    typedef NodeTraits::node_ptr             node_ptr;          //节点指针类型
    typedef NodeTraits::const_node_ptr       const_node_ptr;    //节点常指针类型
    typedef NodeTraits                       node_traits;       //节点特征类

    static void init();                                         //初始化操作
    static bool inited();                                       //是否已经初始化
    static void unlink_after();                                 //断开连接
    static void link_after();                                   //连接
    static void init_header();                                  //初始化头节点
    static std::size_t count();                                 //计算节点的数量
    ...                                                         //其他操作
};
```

使用节点算法我们就可以避免直接操作节点，在更高的抽象层次上实现数据结构。circular_slist_algorithms 可以这样使用：

```
typedef slist_node<void*> node_t;//节点类型，使用 void*来定义指针
typedef slist_node_traits<void*> node_traits_t;                 //节点特征类
typedef circular_slist_algorithms<node_traits_t> algo;          //节点算法

node_t n1, n2;                                                  //两个节点

algo::init_header(&n1);                                         //初始化头节点
```

```
assert(algo::count(&n1) == 1);                    //链表中有一个元素

algo::link_after(&n1, &n2);                       //连接 n1 和 n2
assert(algo::count(&n1) == 2);                    //链表中有两个元素

algo::unlink(&n1);                                //断开 n1 的连接
assert(algo::count(&n2) == 1);                    //链表中有一个元素
```

6.3.4 值特征

值特征（value traits）是一个类似非侵入式容器中值类型 T 的概念，它把用户自有的值类型与节点、节点特征封装在一起，可以提供给节点算法使用。

值特征类通常具有下面的形式：

```
struct value_traits
{
    typedef some_define node_traits;              //节点特征类
    typedef some_define value_type;               //值类型
    typedef some_define node_ptr;                 //节点指针类型
    typedef some_define pointer;                  //指向值的指针类型

    static const link_mode_type link_mode = some_link_mode;   //链接策略

    static node_ptr  to_node_ptr(value_type &value);          //指针转换
    static pointer  to_value_ptr(node_ptr n);                 //指针转换
};
```

值特征类中的一个重要常量是链接模式 link_mode，它是一个枚举值，用于确定侵入式容器的链接处理策略，它有以下三个取值（即三种策略）：

- safe_link ：节点在插入删除时会检查指针，可以地安全插入，是最常用的一种策略；
- auto_unlink：可以在节点对象析构时自动从容器中移除，虽然方便但安全性有所降低，只能用于非常量时间的容器；
- normal_link：节点在插入删除操作时不做任何检查。

这三种链接模式的更详细解说见下一小节。

6.3.5 挂钩

挂钩（hook）是实现侵入式容器的核心概念，它包含节点、节点算法和其他一些信息，我们必须以继承或者成员的方式把挂钩加入自有类（侵入式修改），这样侵入式容器才能通过挂钩用算法操纵节点从而容纳元素。

类摘要

头文件<boost/intrusive/detail/generic_hook.hpp>定义了所有侵入式容器通用的挂钩 generic_hook，类摘要如下：

```
template
   < class GetNodeAlgorithms                      //节点算法元函数
   , class Tag                                    //标记挂钩的标签
   , link_mode_type LinkMode                      //链接策略
   , int HookType                                 //挂钩的类型
   >
class generic_hook
   : public make_default_definer<...>::type
   , public make_node_holder<...>::type
{
   typedef some_define node_algorithms;
   typedef some_define node;
   typedef some_define node_ptr;

public:
   struct boost_intrusive_tags                    //定义一些挂钩的基本属性
   {
      static const int hook_type = HookType;
      static const link_mode_type link_mode = LinkMode;
      typedef Tag  tag;
      typedef some_define node_traits;
      static const bool is_base_hook = some_define;
      static const bool safemode_or_autounlink = some_define;
   };
   bool is_linked() const;                  //挂钩是否已经连接
   void unlink();                           //断开挂钩的连接，要求链接模式为 auto_unlink
};
```

解说

generic_hook 使用了模板元编程来计算类型，把参数转发给工厂元函数 make_default_definer<>和 make_node_holder<>进行计算（多重继承）。

工厂元函数make_default_definer<>使用参数Tag和HookType计算挂钩的类型。HookType 是一个枚举类型，标记挂钩是成员挂钩还是某种基类挂钩。如果是使用缺省标签的基类挂钩，那么计算得到可用于继承的 generic_hook 自身；否则如果是成员挂钩或者自定义标签；那么计算得到一个空类。

工厂元函数 make_node_holder<>使用参数 Tag 计算决定挂钩包含的节点类型。如果是基类挂钩，计算得到一个可用于继承的 node_holder 类型；如果是成员挂钩，那么计算得到节点算法 GetNodeAlgorithms 的节点类型 node。

无论作为基类还是成员，generic_hook 都含有节点和相应的算法，都可以被侵入式容器使用，只是缺省继承基类方式使用时它多了一个内部类型 default_xxx_hook，而成员方式使用时它是一个只具有节点功能的普通类。

generic_hook 有一个重要的成员函数 is_linked()，它可以被节点随时调用，用于检查节点是否已经被链入侵入式容器。

链接模式

generic_hook 模板参数 LinkMode 的取值同值特征类中的定义，它决定了 generic_hook 在构造函数和析构函数时的行为：

- safe_link ：构造时初始化节点为未连接状态，析构时检查节点的连接状态，如果已连接（保存在某个侵入式容器中）则抛出异常；
- auto_unlink：构造时初始化节点为未连接状态，析构时如果已连接则断开连接；
- normal_link：构造、析构时无任何操作。

6.3.6 选项

选项（option）是 intrusive 库中用于配置、调整侵入式容器行为的一大族高阶元数据(参见 11.7.1 小节)。

选项元数据的通常形式是：

```
template<class OptionName>
```

```
struct option_name                          //选项名
{
    template<class Base>
    struct pack : Base                      //继承
    {
        typedef OptionName option_name;     //类型定义
    };
};
```

元数据 option_name 内部的元函数是 pack<>，它会把参数 OptionName 定义为一个与元数据同名的类型（即 option_name）。使用 park<>而不是元编程中更常用的 apply<>的原因很简单，因为这些高阶元数据要配合另一个元函数 pack_options<>来执行元数据打包的操作①。

pack_options 支持最多 11 个选项元数据，因为使用了元编程，所以不必用固定的顺序指定参数，它的简化形式如下：

```
template<class Prev, class Next>
struct do_pack                         //子元函数，调用高阶元数据的 park<>元函数
{
   typedef typename Next::template pack<Prev> type;
};
template< class DefaultOptions, class O1, class O2 >
struct pack_options
{
  typedef
  typename do_pack
    < typename do_pack
      < DefaultOptions , O1 >::type  //对 O1 打包
    , O2                             //对 O2 打包
    >::type type;
};
```

pack_options 连续调用元函数 do_pack<>对所有选项逐个命名，中间又有元数据的继承，所以最后得到的类型就是一个含有 N 个具有小写同名类型定义的大类型，把选项打包到了一起。

为了方便使用选项，intrusive 库定义了一个默认的选项集合 hook_defaults，它包含了大部分常用的选项：

① 当然使用 mpl 中的惯例 apply<>也是可以的，大概库作者认为使用 pack<>的意义会更明确吧。

```
struct hook_defaults
   : public pack_options
      < none                        //打包的结束点
      , void_pointer<void*>         //使用 void*定义指针
      , link_mode<safe_link>        //安全链接模式
      , tag<default_tag>            //默认标签
      , optimize_size<false>        //不优化空间，优化时间
      , store_hash<false>           //无序容器专用，散列值存储在外部
      , linear<false>               //非线性化，使用循环链接
      , optimize_multikey<false>    //无序容器专用，不优化重复键值的元素
      >::type
{};
```

6.3.7 处置器

因为侵入式容器不做内存管理，不能销毁元素，所以 intrusive 库提出了处置器（Disposer）的概念来帮助侵入式容器“销毁”元素。

处置器是一个函数对象，它可以用某种策略处理元素指针，形如：

```
struct Disposer                              //处置器函数对象
{
   void operator()(T *to_dispose)            //operator()接受指针
   { ...}// 处置指针，通常是 delete to_dispose 来销毁对象
};
```

侵入式容器的 pop_back()、clear()、erase()等涉及容器内元素移除的操作都有两个版本。第一个版本是与标准容器相同的接口，只是简单地断开元素在容器中的链接，元素没有被真正销毁，用户必须在适当的时候管理这些移出容器的元素，这通常是一个很困难的事情（如果使用指针容器则不会有这样的困扰）。第二个版本是带“_and_dispose”版本的函数，它除了标准参数外还多出一个处置器参数，侵入式容器使用它来处置对象。

6.3.8 克隆

侵入式容器没有内存管理功能，因此是不可赋值和不可拷贝的，但与指针容器类似，它提供了克隆的概念，允许一个容器从另一个容器克隆元素。

所有的侵入式容器都提供一个 clone_from()成员函数，它实现克隆操作，基本形式

如下：

```
template <class Cloner, class Disposer>
void clone_from(const Cont &src, Cloner c, Disposer d);
```

两个模板参数分别是克隆器和处置器函数对象，容器先使用处置器 d 清除原有的所有元素，然后使用克隆器 c 从 src 逐个克隆元素到本容器，如果克隆过程中发生异常则调用处置器 d 清除已经克隆的元素。

克隆器是一个函数对象，它的形式是：

```
struct cloner
{
   template<typename ValueType>
   ValueType* operator()(const ValueType& r);
};
```

克隆器的 operator()接受一个元素类型的常引用，然后以指针的形式返回它的克隆对象。

6.4 链表

intrusive 库提供两种链表式侵入式容器：slist 和 list。我们已经在 6.2 小节见到了 slist 的部分用法，它是单链表，只有一个指针的空间开销，但时间复杂度较高，因此在这里我们介绍更常用的双向链表 list，它有两个指针的空间开销，类似 std::list，用法更灵活。

list 位于名字空间 boost::intrusive，需要包含头文件<boost/intrusive/list.hpp>，即：

```
#include <boost/intrusive/list.hpp>
using namespace boost::intrusive;
```

6.4.1 节点和算法

list 使用的节点是 list_node，它提供了前驱和后继两个指针：

```
template<class VoidPointer>
struct list_node
{
```

```
    typedef typename boost::pointer_to_other
      <VoidPointer, list_node>::type node_ptr;          //指针类型定义
    node_ptr next_;                                       //前驱指针
    node_ptr prev_;                                       //后继指针
};
```

list_node 对应的节点特征类是 list_node_traits，封装了对 list_node 的所有操作：

```
template<class VoidPointer>
struct list_node_traits
{
    typedef list_node<VoidPointer> node;
    typedef some_define  node_ptr;
    typedef some_define const_node_ptr;

    static node_ptr get_previous(const_node_ptr n);
    static void set_previous(node_ptr n, node_ptr prev);
    static node_ptr get_next(const_node_ptr n);
    static void set_next(node_ptr n, node_ptr next);
};
```

list 使用的节点算法是 circular_list_algorithms，接口与 6.3.3 小节的 circular_slist_algorithms 类似。

6.4.2 基类挂钩

list 使用的基类挂钩是 list_base_hook，它的类摘要如下：

```
template<class O1, class O2, class O3>
class list_base_hook
    : public make_list_base_hook<O1, O2, O3>::type
{
  void swap_nodes(list_base_hook &);
  bool is_linked() const;              //挂钩是否已经连接
  void unlink();                       //断开挂钩的连接，要求链接模式为 auto_unlink
}
```

list_base_hook 派生自 generic_hook（6.3.5 小节），实际上是工厂元函数 make_list_base_hook<>的计算结果：

```
template<class O1 = none, class O2 = none, class O3 = none>
```

```
struct make_list_base_hook
{
   typedef typename pack_options          //选项元数据打包
     < hook_defaults,                     //使用缺省挂钩参数
     O1, O2, O3 >::type packed_options;

   typedef detail::generic_hook           //使用挂钩基类
   < get_list_node_algo<...>              //使用 circular_list_algorithms
   , typename packed_options::tag         //标签
   , packed_options::link_mode            //链接模式
   , detail::ListBaseHook                 //链表基类挂钩枚举值
   > implementation_defined;

   typedef implementation_defined type;   //返回元函数计算结果
};
```

list_base_hook 可以使用如下三个选项，因为使用了选项元数据打包，所以对顺序无要求：

- tag<> ：一个语法层面的标签，用于区分不同类别的挂钩。不同的类可以使用相同的标签，但同一个类如果使用多个基类挂钩那么标签不能相同，默认值是 default_tag；
- void_pointer<>：挂钩使用的指针类型，默认值是 void*；
- link_mode<> ：挂钩的链接模式，默认值是 safe_link。

一个完整定义的链表基类挂钩可以是这样：

```
list_base_hook<
         tag<struct some_tag>,            //使用一个不完整类型定义标签
         void_pointer<void*>,             //指针通常都使用 void*定义
         link_mode<safe_link> >           //链接模式使用 safe_link
```

6.4.3 成员挂钩

list 使用的成员挂钩是 list_member_hook，它的类摘要如下：

```
template<class O1, class O2, class O3>
class list_member_hook
   : public make_list_member_hook
     <O1, O2, O3>::type
{...};                                    //接口同 list_base_hook
```

list_member_hook 也是由一个工厂元函数 make_list_member_hook<>产生的，与 make_list_base_hook<>仅有微小的不同。

```
template<class O1 = none, class O2 = none, class O3 = none>
struct make_list_member_hook
{
  ...//省略相同的代码

   typedef detail::generic_hook                    //使用挂钩基类
   < get_list_node_algo<...>                       //使用 circular_list_algorithms
   , member_tag                                    //成员标签
   , packed_options::link_mode                     //链接模式
   , detail::NoBaseHook                            //成员（非基类）挂钩类型
   > implementation_defined;
};
```

list_member_hook 仅仅是在挂钩类别上与 list_base_hook 不同，模板参数和成员函数均相同。

6.4.4　list 类摘要

双向链表容器 list 的定义同样也使用了复杂的元编程计算，它的真正实现是模板类 list_impl，元函数 make_list<>根据模板参数计算值特征等配置选项，由于其元计算过程较复杂，本书只给出 list 的接口定义：

```
template<class T, class O1, class O2, class O3 >
class list {
public:
  // 容器必需的若干类型定义
  typedef some_define value_traits;
  typedef some_define value_type;
  ...//其他类型定义

  //构造函数
  list();
  template<typename Iterator> list(Iterator, Iterator);

  //双向链表的通用操作
  void push_back(reference);
  void push_front(reference);
  void pop_back();
  void pop_front();
  ...//其他迭代器、容量、删除、连接、排序、逆序、合并等操作，同标准链表容器
```

```
    //左右移动算法
    void shift_backwards(size_type = 1) ;
    void shift_forward(size_type = 1) ;

    //下面是使用处置器的操作，为代码简单起见忽略了模板参数列表
    void pop_back_and_dispose(Disposer) ;
    void pop_front_and_dispose(Disposer) ;

    iterator erase_and_dispose(iterator, Disposer) ;
    iterator erase_and_dispose(iterator, iterator, Disposer) ;

    void clear_and_dispose(Disposer) ;

    void dispose_and_assign(Disposer, Iterator, Iterator) ;

    void remove_and_dispose(const_reference, Disposer) ;
    void remove_and_dispose_if(Pred, Disposer) ;

    void unique_and_dispose(Disposer) ;
    void unique_and_dispose(BinaryPredicate, Disposer) ;

    //克隆
    void clone_from(const list &, Cloner, Disposer) ;

    //从值获得迭代器
    iterator iterator_to(reference) ;
    static iterator s_iterator_to(reference) ;

    // 迭代器到容器的静态成员函数
    static list & container_from_end_iterator(iterator) ;
  };
  ...//各种比较操作符定义，如==、!=、<
```

list 有以下四个模板参数，第一个参数 T 是容器的元素类型，其他三个是配置选项，其中两个通常使用缺省值，无须变动：

- constant_time_size<bool Enabled>：是否使用一个额外的变量保存容器的大小，这样可以在常量时间里获得容器的容量信息，默认是 true。如果挂钩的链接模式使用 auto_unlink，那么它必须被置为 false；

- size_type<class SizeType> ：容器大小的类型，默认是 std::size_t。

下面的三个选项标明了容器的值特征，只能选择一个使用；

- base_hook<typename BaseHook>：容器使用基类挂钩，指定基类挂钩类型，值特征被自动推导，是 list 的默认配置；
- member_hook<typename Parent, typename MemberHook, MemberHook Parent::* PtrToMember>：容器使用成员挂钩，需要在模板参数中明确元素类型、成员挂钩类型和挂钩的成员变量访问指针，值特征被自动推导；
- value_traits<typename ValueTraits>：用户手工指定值特征。

工厂元函数 make_list<>可以使用同样的模板参数得到链表容器 list，它的编译速度更快，而且能够保证使用不同的选项顺序都能够创建出相同的链表类型。

6.4.5 list 的基本用法

使用 list 容纳元素必须先使用基类挂钩 list_base_hook 或者成员挂钩 list_member_hook 侵入式修改元素的定义，通常我们无须修改挂钩的选项。

在定义 list 时我们最好使用工厂元函数 make_list<>，必需的参数是容纳的元素类型。基类挂钩 base_hook<>是被默认使用的，但如果挂钩没有使用缺省标签而是自定义标签那么必须明确写出，如果元素使用了成员挂钩就需要用 member_hook<>指定挂钩的类型和访问方式。

list 完全模仿了 std::list 的接口，符合标准容器的定义，基本上 std::list 的所有操作都适用于 list，但需要注意 list 不支持拷贝构造和赋值。

为了示范 list 的用法，我们给 point 类增加比较操作符定义：

```
#include <boost/intrusive/list.hpp>
using namespace boost::intrusive;

class point: public list_base_hook<>                    //使用基类挂钩
{
public:
  ..//数据和构造函数定义同前
  friend bool operator==(const point& l, const point& r)     //相等定义
```

```
  {   return l.x == r.x && l.y == r.y;}
  friend bool operator<(const point& l, const point& r)      //小于定义
  {   return l.x < r.x;   }
};
```

示范 list 基本操作的代码如下：

```
int main()
{
  using namespace boost::assign;               //assign 名字空间
  ptr_vector<point> vec =                      //指针容器
      ptr_list_of<point>(0)(1)(2)(3)(4);       //使用 assign 库初始化

  typedef make_list<point>::type list_t;       //使用工厂元函数，基类挂钩
  list_t lt;                                   //声明链表侵入式容器
  assert(lt.empty());                          //此时容器为空

  lt.push_back(vec[2]);                        //向末端添加元素，链表是[2]
  lt.push_front(vec[3]);                       //向前端添加元素，链表是[3, 2]

  assert(!lt.empty() && lt.size() == 2);       //获得容器的大小
  assert(lt.front().x == 3);                   //访问前端元素
  assert(lt.back().x == 2);                    //访问末端元素

  lt.insert(boost::next(lt.begin()),           //在第二个位置执行插入操作
      vec.begin(), vec.begin() + 2);           //插入一个区间内的所有元素
  BOOST_FOREACH(point& p, lt)                  //使用 foreach 算法遍历
  {
      cout << p.x << ","; //输出 3, 0, 1, 2
  }

  lt.reverse();                                //逆序链表，链表是[2, 1, 0, 3]
  lt.pop_front();                              //弹出前端元素，链表是[1, 0, 3]
  assert(lt.size() == 3);

  lt.insert(boost::prior(lt.end()), vec[4]); //在末端前插入一个元素
                                               //链表是[1, 0, 4, 3]

  lt.sort();                                   //排序，链表是[0, 1, 3, 4]
```

```
    //删除元素，链表是[0，1，4]
    lt.erase(boost::prior(lt.end(), 2));        //注意 prior()前进了两步
}
```

基类挂钩使用自定义标签时代码需要做少量的变动，因为这时无法自动推导出基类挂钩类型，必须使用 base_hook<>明确地写出挂钩类型：

```
class point: public list_base_hook<tag<struct a_tag> >{};    //自定义标签
typedef make_list<point,
        base_hook<list_base_hook<tag<a_tag>>> >::type list_t;
```

成员挂钩的用法与基类挂钩没有太大的差别，只是必须在 list 的模板参数列表中配置 member_hook<>选项，略显麻烦，例如：

```
class point                                         //无继承
{
public:
    ...//同前
    list_member_hook<> m_hook;                      //成员挂钩，缺省配置
};
typedef make_list<point,
                  member_hook<point,                //成员挂钩选项
                  list_member_hook<>, &point::m_hook>>::type list_t;
```

6.4.6 list 的特有用法

list 拥有一些不同于 std::list 的特有用法，本小节将逐个详解这些接口。

左右移动

成员函数 shift_backwards()和 shift_forward()相当于 std::rotate 算法，令链表中的元素循环后移或前移，默认移动一个元素。例如：

```
using namespace boost::assign;              //assign 名字空间
ptr_vector<point> vec =                     //指针容器
    ptr_list_of<point>(0)(1)(2)(3)(4);      //使用 assign 库初始化

typedef make_list<point>::type list_t;      //使用基类挂钩
list_t lt(vec.begin(), vec.end());          //区间元素构造
assert(lt.size() == 5);                     //5 个元素，链表是[0，1，2，3，4]
```

```
//循环后移两个元素，链表是[3，4，0，1，2]
lt.shift_backwards(2);
//循环前移一个元素，链表是[4，0，1，2，3]
lt.shift_forward();
```

处置器相关操作

list 为 pop_back()、pop_front()、clear()、erase()、remove()、assign() 和 unique() 等涉及容器内元素移除的操作提供了对应的使用处置器的版本，缀词“dispose”的位置表示了处置器的使用时机，大部分函数都是操作完毕后使用处置器的 operator() 处理移除的元素，只有 dispose_and_assign() 是先使用处置器然后再赋值。

作为示例我们先定义一个简单的处置器函数对象，它输出处置的值然后删除之：

```
struct disposer                                //一个简单的处置器
{
  void operator()(point* p)                    //操作指针
  {
      cout << "dispose:" << p->x << endl;      //输出值
      checked_delete(p);                       //使用 checked_delete，更加安全
  }
};
```

处置器接口的示范代码如下：

```
std::vector<point*> vec;                        //一个容纳原始指针的标准容器
for (int i = 0;i < 5; ++i)                      //添加 5 个指针
{
    vec.push_back(new point(i, i));
}

typedef make_list<point>::type list_t;          //侵入式容器
list_t lt(make_indirect_iterator(vec.begin()),  //迭代器区间构造
    make_indirect_iterator(vec.end()));         //使用间接迭代器，见 3.6.4 小节

disposer d;                                     //一个处置器对象

lt.pop_front_and_dispose(d);                    //弹出前端元素

lt.erase_and_dispose(boost::next(lt.begin()), d);  //删除第二个元素
```

```
lt.remove_and_dispose(point(4, 4), d);              //移除值为(4，4)的元素
lt.push_back(*(new point(3, 3)));                   //末端增加一个元素

lt.unique_and_dispose(d);                           //移除重复元素
```

程序的运行结果如下：

```
//初始链表元素是[0，1，2，3，4]
dispose:0       //链表是 [1，2，3，4]
dispose:2       //链表是[1，3，4]
dispose:4       //链表是[1，3]
 //插入一个元素，链表是[1，3，3]
dispose:3       //链表是[1，3]
```

处置器大多数情况下执行删除指针操作，但它也可以是其他任意的操作，比如把指针移动到一个指针容器中。例如，把处置器修改如下：

```
struct disposer
{
  template<typename Cont>                         //模板参数是一个序列指针容器
  void operator()(point* p, Cont* c)              //成员模板函数，双参数
  {
      c->push_back(p);                            //把指针移动到指针容器中
  }
};
```

这个处置器可以使用 bind 绑定使用的指针容器，把它适配为侵入式容器可以使用的单参数版本：

```
disposer d;                                     //处置器对象
ptr_vector<point> pvec;                         //一个指针容器
lt.pop_front_and_dispose(
      boost::bind<void>(d, _1, &pvec));         //bind 绑定指针容器
assert(pvec.size() == 1);                       //处置后指针被移至指针容器
```

代码中的 bind 需显式用模板参数指明返回类型（void），这是因为处置器函数对象 disposer 没有内部类型定义 result_type。

克隆

成员函数 clone_from()使用侵入式容器的克隆概念（6.3.8 小节）从另一个容器中

克隆元素到本容器，要求元素必须有克隆器和处置器两个函数对象。

对于支持拷贝构造的类型来说，泛用的克隆器函数对象可以实现如下：

```
struct cloner                                       //泛用的克隆器函数对象
{
  template<typename T>
  T* operator()(const T& r)
  {   return factory<T*>()(r);}                     //使用 factory 函数对象代替 new
};
```

示范 clone_from()用法的代码如下：

```
ptr_vector<point> vec =                             //指针容器
  ptr_list_of<point>(0)(1)(2)(3)(4);                //使用 assign 库初始化

typedef make_list<point>::type list_t;
list_t lt(vec.begin(), vec.end());                  //迭代器区间构造

list_t lt2;                                         //另一个侵入式链表容器
assert(lt2.empty());                                //此时容器无元素

lt2.clone_from(lt, cloner(), disposer());           //从 lt 克隆元素
assert(lt2 == lt);//比较两个容器内的元素，必定相等
```

迭代器特殊操作

因为侵入式容器修改了元素的代码，所有的元素都含有链接信息，所以侵入式容器可以直接从一个值获得对应的迭代器。这个功能要求元素必须已经被容器容纳，即挂钩的is_linked()返回 true，否则会产生一个断言异常。

所有的侵入式容器都提供成员函数 iterator_to()，它可以返回元素对应的迭代器位置，大多数侵入式容器（无序容器除外）还提供一个同等功能的静态成员函数s_iterator_to()。

更进一步，静态成员函数 container_from_end_iterator()还可以从逾尾迭代器获得侵入式容器的引用。

示范这些成员函数用法的代码如下：

```
ptr_vector<point> vec =                             //指针容器
  ptr_list_of<point>(0)(1)(2)(3)(4);                //使用 assign 库初始化

typedef make_list<point>::type list_t;              //侵入式链表容器
```

```
list_t lt;

lt.push_back(vec[1]);                               //添加两个元素
lt.push_back(vec[3]);

assert(vec[1].is_linked());                         //元素已经被侵入式容器容纳
assert(lt.iterator_to(vec[1]) == lt.begin());       //获得迭代器

assert(vec[3].is_linked());                         //元素已经被侵入式容器容纳
assert(list_t::s_iterator_to(vec[3]) ==             //获得迭代器
  boost::prior(lt.end()));                          //是逾尾迭代器之前的位置

//从逾尾迭代器得到容器的引用
list_t& rlt = list_t::container_from_end_iterator(lt.end());
assert(addressof(rlt) == addressof(lt));            //两者是同一个对象
```

6.5 有序集合

intrusive 库基于红黑树、AVL 树、splay 树、scapegoat 树、二叉树和堆实现了五种侵入式有序集合容器，这些集合容器除了内部使用的算法不同外接口基本相同，都类似标准集合容器，故本书仅介绍基于红黑树的 set 和 multiset（下文中的“有序集合容器”一词均指这两个容器），其他容器读者可举一反三。

set 和 multiset 位于名字空间 boost::intrusive，需要包含头文件 <boost/intrusive/set.hpp>，即：

```
#include <boost/intrusive/set.hpp>
using namespace boost::intrusive;
```

6.5.1 节点和算法

有序集合容器基于红黑树实现，使用的节点类名字是 rbtree_node。为了使用 optimize_size<>选项决定优化策略它有两个实现类，紧凑版本可以节省一个整数的空间：

```
template<class VoidPointer>
struct compact_rbtree_node                          //紧凑版本
{
```

```
    typedef typename pointer_to_other
      <VoidPointer,compact_rbtree_node<VoidPointer> >::type node_ptr;
    enum color { red_t, black_t };          //颜色枚举
    node_ptr parent_, left_, right_;        //父指针和左右指针
};
struct rbtree_node                          /普通版本
{
    ...//同 compact_rbtree_node
    color color_;                           //多出的颜色变量
};
```

节点特征类 rbtree_node_traits 使用一个 bool 类型模板参数 OptimizeSize 定制，它再使用元函数 rbtree_node_traits_dispatch<>特化到具体的实现类 default_rbtree_node_traits_impl<VoidPointer>或者 compact_rbtree_node_traits_impl<VoidPointer>:

```
template<class VoidPointer, bool OptimizeSize = false>
struct rbtree_node_traits
    :public rbtree_node_traits_dispatch<VoidPointer, OptimizeSize>
{};
```

有序集合容器使用的算法是 rbtree_algorithms，接口较链表容器的算法要复杂很多，本书从略。

6.5.2 基类挂钩

有序集合容器使用的基类挂钩是 set_base_hook，它的类摘要如下：

```
template<class O1, class O2, class O3, class O4>
class set_base_hook
    :  public make_set_base_hook<O1, O2, O3, O4>::type
{...};//成员同 list_base_hook
```

set_base_hook 同样使用了工厂元函数，核心代码如下：

```
template<class O1, class O2, class O3, class O4>
struct make_set_base_hook
{
    typedef detail::generic_hook    //使用挂钩基类
    < get_set_node_algo<...>        //使用 rbtree_algorithms
```

```
    , typename packed_options::tag          //标签
    , packed_options::link_mode             //链接模式
    , detail::SetBaseHook                   //有序集合基类挂钩枚举值
    > implementation_defined;
};
```

set_base_hook 可以使四个选项：tag<>、void_pointer<>、link_mode<>和 optimize_size<>，其中前三个的含义同 list_base_hook（6.4.2 小节），而第四个 optimize_size<>可以取值 true 或 false，决定集合容器是优化空间还是优化时间。

6.5.3 成员挂钩

有序集合容器使用的成员挂钩是 set_member_hook，它的类摘要如下：

```
template<class O1, class O2, class O3, class O4>
class set_member_hook
   : public make_set_member_hook<O1, O2, O3, O4>::type
{...};
```

set_member_hook 使用的工厂元函数 make_set_member_hook<>核心代码如下：

```
template<class O1, class O2, class O3, class O4e>
struct make_set_member_hook
{
    typedef detail::generic_hook>            //使用挂钩基类
    < get_set_node_algo<...>                 //使用 rbtree_algorithms
    , member_tag                             //成员标签
    , packed_options::link_mode              //链接模式
    , detail::NoBaseHook                     //成员挂钩枚举值
    > implementation_defined;
};
```

set_member_hook 仅仅是在挂钩类别上与 set_base_hook 不同，模板参数、成员函数均相同。

6.5.4 set 类摘要

set 与 std::set 类似，容纳不允许重复的元素，它的类摘要如下：

```
template<class T, class O1, class O2, class O3, class O4>
class set
   : public make_set<T, O1, O2, O3, O4>::type    //工厂元函数
{
```

```
public:
...//类型定义和与 std::set 相同的操作

//构造函数
set(const value_compare & );
set(Iterator, Iterator, const value_compare &);

//克隆
void clone_from(const set &, Cloner, Disposer) ;

//使用键操作
size_type erase(const KeyType &, KeyValueCompare) ;
size_type erase_and_dispose(const KeyType &, KeyValueCompare, Disposer) ;
size_type count(const KeyType &, KeyValueCompare) const;
iterator lower_bound(const KeyType &, KeyValueCompare) ;
iterator upper_bound(const KeyType &, KeyValueCompare) ;
iterator find(const KeyType &, KeyValueCompare) ;
std::pair< iterator, iterator >
  equal_range(const KeyType &, KeyValueCompare) ;

//带检查的插入操作
std::pair< iterator, bool >
insert_check(const KeyType &, KeyValueCompare,insert_commit_data &);
std::pair< iterator, bool >
insert_check(const_iterator, const KeyType &, KeyValueCompare,
             insert_commit_data &) ;
iterator insert_commit(reference, const insert_commit_data &) ;

//使用处置器的操作
iterator erase_and_dispose(iterator, Disposer) ;
iterator erase_and_dispose(iterator, iterator, Disposer) ;
size_type erase_and_dispose(const_reference, Disposer) ;
size_type erase_and_dispose(const KeyType &, KeyValueCompare,
              Disposer) ;
void clear_and_dispose(Disposer) ;

//从值获得迭代器
iterator iterator_to(reference) ;
static iterator s_iterator_to(reference) ;

// 迭代器到容器的静态成员函数
static set & container_from_end_iterator(iterator) ;
```

```
  static set & container_from_iterator(iterator) ;
};
```

set 使用四个模板参数，其中的 base_hook<>/member_hook<>/value_traits<>、constant_time_size<>和 size_type<>与 list 的含义相同，compare<>选项相当于 std::set 的比较谓词参数，用于定制排序准则，缺省是 std::less<T>。

工厂元函数 make_set<>同样用于生产 set 容器，我们应该常使用它。

6.5.5 set 的基本用法

set 具有与 std::set 相同的接口，因而很容易使用，我们只需要设置好元素类的挂钩，然后就可以使用 set 了，需要注意的是比较谓词应该使用 compare<>来配置，这一点与 std::set 不同。

为了使用 set 容器，需要修改 point 的代码，使用 set_base_hook 或者 set_member_hook，例如，基类挂钩的实现如下：

```
class point: public set_base_hook<>,      //set 基类挂钩
boost::less_than_comparable<point>        //使用 operators 库，增加其他操作符
{...};                                    //同前
```

示范 set 基本用法的代码如下：

```
#include <boost/intrusive/set.hpp>
using namespace boost::intrusive;

int main()
{
  using namespace boost::assign;                      //assign 名字空间
  ptr_vector<point> vec =                             //指针容器
      ptr_list_of<point>(0)(1)(2)(3)(4);              //使用 assign 库初始化

  typedef make_set<point,                             //使用工厂元函数
      compare<std::greater<point>> >::type set_t;     //设置大于比较谓词
  set_t s;

  assert(s.empty());                                  //初始容器为空

  assert(s.insert(vec[0]).second);                    //插入一个元素
  assert(s.insert(vec[2]).second);
```

```
    assert(!s.insert(vec[2]).second);                    //不允许重复插入

    assert(s.size() == 2);                               //获得容器大小
    assert(s.count(point()) == 1);                       //计算元素的个数

    s.insert(vec.begin(), vec.end());            //插入一个区间内的元素，重复的不插入
    assert(s.size() == 5);

    s.erase(s.lower_bound(2), s.upper_bound(2));         //删除元素
}
```

set 的用法比较简单，代码中需要注意的是我们使用了 compare<>选项，用 std::greater 而不是 std::less 作为比较谓词，因此这个有序集合是降序的。

6.5.6 set 的特有用法

set 具有与 list 相同的一些侵入式容器特有接口，包括克隆、处置器、迭代器的特殊操作，比较特别的是它可以无须使用逾尾迭代器就可以从迭代器获得容器的引用。示范这些特殊操作用法的代码如下：

```
std::vector<point*> vec;                         //一个容纳原始指针的标准容器
for (int i = 0;i < 5; ++i)                       //添加 5 个指针
{
    vec.push_back(new point(i, i));
}

typedef make_set<point>::type set_t;             //有序集合侵入式容器，升序
set_t s(make_indirect_iterator(vec.begin()),     //迭代器区间构造
    make_indirect_iterator(vec.end()));          //使用间接迭代器

//删除容器中的前两个元素
s.erase_and_dispose(s.begin(), boost::next(s.begin(), 2),
    disposer());

set_t s2;
s2.clone_from(s, cloner(), disposer());          //克隆容器
assert(s2.begin()->x == 2);
assert(s2 == s);                                 //比较两个容器，应相等

//从一个任意迭代器获得容器的引用
```

```
assert(addressof(s2) ==
  addressof(s2.container_from_iterator(boost::next(s2.begin()))));
```

使用键操作值

有的时候容器内存储的值类型（value_type）构造成本很高，为了避免使用大临时对象带来的开销，set 允许使用一个函数对象 KeyValueCompare 比较容器中元素是否与一个低成本的键 KeyType 相等，从而提高了运行效率。

函数对象 KeyValueCompare 是一个不等关系比较，它应该是与集合容器 compare<> 选项配置的排序准则谓词一致的（简单地说就是同为小于关系或者同为大于关系）。

我们可以为 point 定义一个小于关系的键比较函数对象，它与 std::less<point> 一致：

```
struct key_compare
{
  typedef const int& key_type;                    //键类型
  typedef const point& value_type;                //值类型

  //因为要比较不同类型，所以必须有两个 operator()函数
  bool operator()(key_type k, value_type p) const
  {   return k < p.x;}
  bool operator()(value_type p, key_type k) const
  {   return p.x < k;}
};
```

set 的 count()、find()、erase()等成员函数都有使用键的重载形式，使用 key_compare 函数对象的代码如下：

```
ptr_vector<point> vec =                           //指针容器
  ptr_list_of<point>(0)(1)(2)(3)(4);              //使用 assign 库初始化

typedef make_set<point>::type set_t;              //使用缺省的 std::less<>
set_t s(vec.begin(), vec.end());                  //区间构造

key_compare kc;                                   //一个键比较函数对象

assert(s.count(1, kc) == 1);                      //使用键计算元素数量
assert(s.find(2, kc)->x == 2);                    //使用键查找元素
assert(s.find(9, kc) == s.end());
```

```
assert(s.erase(3, key_compare()) == 1);   //使用键删除元素
assert(s.find(point(3)) == s.end());      //构造值对象查找，开销大
```

“键-值”的功能除了用于处理大对象，也可以用来模拟实现映射容器，读者可自行试验。

检查插入

set 使用键操作为插入提供了一套类似数据库的 check-commit 机制，它可以用在值类型的构造成本很高的场合。insert_check()函数使用 KeyValueCompare 比较容器中元素是否与 KeyType 相等，避免了构造大对象的开销，如果允许插入（返回的 pair 的 second 成员为 true），那么就可以紧接着调用 insert_commit()来完成插入操作。

insert_check()和 insert_commit()特别适合于侵入式容器容纳大对象的场合，在这里我们仅用 point 类做个示范：

```
ptr_vector<point> vec =                          //指针容器
  ptr_list_of<point>(0)(1)(2)(3)(4);             //使用 assign 库初始化

typedef make_set<point>::type set_t              //使用缺省的 std::less<>
set_t s(vec.begin(),boost::next(vec.begin(), 3)); //插入三个元素

set_t::insert_commit_data idata;                 //一个用于提交的数据类型

//比较键 0，已经存在，无法插入
assert(!s.insert_check(0, key_compare(), idata).second);

//比较键 4，不存在，可以插入
assert(s.insert_check(4, key_compare(), idata).second);
s.insert_commit(vec[4], idata);

assert(s.find(4, key_compare()) != s.end())      //4 已经插入容器
```

6.5.7 multiset 类摘要

multiset 与 std::multiset 类似，可以容纳任意的元素，允许重复，它的类摘要如下：

```
template<class T, class ...Options>
```

```
class multiset
    : public make_multiset<T,O1, O2, O3, O4>::type
{...};//基本同 set
```

multiset 的接口与 set 基本相同，但因为它允许插入重复的元素，所以没有 insert_check()和 insert_commit()系列函数。

工厂元函数 make_multiset<>用于生产 multiset 容器。

6.5.8 multiset 的用法

multiset 的用法与 set 几乎是一样的，它们的唯一区别就是是否允许重复的元素。

简单示范 multiset 用法的代码如下：

```
class point
{
  ...//同前
  set_member_hook<> m_hook;                         //使用成员挂钩
};

int main()
{
  ptr_vector<point> vec =                           //指针容器
   ptr_list_of<point>(0)(1)(2)(3)(4)(3)(2);         //使用 assign 库初始化

  //定义 multiset
  typedef make_multiset<point,
   member_hook<point, set_member_hook<>, &point::m_hook>>::type set_t;
  set_t s(vec.begin(),vec.end());                   //区间构造
  assert(s.size() == 7);

  s.erase(2, key_compare());                        //使用键删除元素
  assert(s.size() == 5);
  assert(s.count(2, key_compare()) == 0);           //使用键计算元素个数
}
```

6.6 无序集合

unordered_set 和 unordered_multiset 是无序集合容器，它们不是使用指针链接而是使用散列表来实现元素的查找与存储，因而其实现方式比较特殊。

unordered_set 和 unordered_multiset 位于名字空间 boost::intrusive，需要包含头文件<boost/intrusive/unordered_set.hpp>，即

```
#include <boost/intrusive/unordered_set.hpp>
using namespace boost::intrusive;
```

6.6.1 节点和算法

无序集合容器使用的节点类是 unordered_node，它基于单链表节点 slist_node 实现，并使用 bool 类型模板参数进行了特化，摘要如下：

```
template<class VoidPointer, bool StoreHash, bool OptimizeMultiKey>
struct unordered_node
   : public slist_node<VoidPointer>
{
   typedef some_define    node_ptr;
   node_ptr               prev_in_group_; //根据 OptimizeMultiKey 选项可被优化
   std::size_t hash_;                     //根据 StoreHash 选项可被优化
};
```

无序集合容器的节点特征类是 unordered_node_traits，它基于单链表特征类，增加了操作散列值的成员函数：

```
template<class VoidPointer, bool StoreHash, bool OptimizeMultiKey>
struct unordered_node_traits
   : public slist_node_traits<VoidPointer>
{
   static std::size_t get_hash(const node_ptr n);
   static void set_hash(node_ptr n, std::size_t h);
};
```

无序集合容器使用的算法是 unordered_algorithms，它基于 circular_slist_algorithms。

6.6.2 基类挂钩

无序集合容器使用的基类挂钩是 unordered_set_base_hook，它的类摘要如下：

```
template<class O1, class O2, class O3, class O4>
class unordered_set_base_hook
   : public make_unordered_set_base_hook< O1, O2, O3, O4>::type
```

```
{...};//成员同 list_base_hook
```

工厂元函数 make_unordered_set_base_hook<>的核心代码如下：

```
template<class O1, class O2, class O3, class O4>
struct make_unordered_set_base_hook
{
   typedef detail::generic_hook             //使用挂钩基类
   < get_uset_node_algo<...>                //使用 unordered_algorithms
   , typename packed_options::tag           //标签
   , packed_options::link_mode              //链接模式
   , detail::UsetBaseHook                   //有序集合基类挂钩枚举值
   > implementation_defined;
};
```

unordered_set_base_hook 可以使用五个选项：tag<>、void_pointer<>、link_mode<>、store_hash<>和 optimize_multikey<>，其中前三个的含义同 list_base_hook（6.4.2 小节），后两个选项的含义如下：

- store_hash<>：挂钩（节点）中保存散列值（hash_），重散列时无须重新计算，可以提高性能，缺省值为 false；
- optimize_multikey<>：unordered_multiset 专用的选项，挂钩（节点）中存储指针（prev_in_group_），可以把相等的元素聚集在一起提高性能，缺省值为 false。

6.6.3 成员挂钩

无序集合容器使用的成员挂钩是 unordered_set_member_hook，它的类摘要如下：

```
template<class O1, class O2, class O3, class O4>
class unordered_set_member_hook
   : public make_unordered_set_member_hook< O1, O2, O3, O4>::type
{...};//成员同 list_base_hook
```

工厂元函数 make_unordered_set_member_hook<>的核心代码如下：

```
template<class O1, class O2, class O3, class O4>
struct make_unordered_set_member_hook
{
```

```
    typedef detail::generic_hook              //使用挂钩基类
    < get_uset_node_algo<...>                 //使用 unordered_algorithms
    , member_tag                              //成员挂钩标签
    , packed_options::link_mode               //链接模式
    , detail::NoBaseHook                      //成员挂钩枚举值
    > implementation_defined;
};
```

unordered_set_member_hook 仅仅是在挂钩类别上与 unordered_set_base_hook 不同，模板参数、成员函数均相同。

6.6.4 unordered_set 类摘要

unordered_set 类似无序容器 boost::unordered_set 或者 std::tr1::unordered_set，类摘要如下[①]：

```
template<class T, class O1, ..., class O10>
class unordered_set
   : public make_unordered_set<T,...>::type         //工厂元函数
{
  public:
  typedef some_define bucket_type;                   //桶类型
  typedef some_define bucket_traits;                 //桶特征
  ...//其他类型定义和与 boost::unordered_set 相同的操作

  //构造函数
  unordered_set(const bucket_traits & );
  unordered_set(Iterator, Iterator, const bucket_traits &);

  //散列容器专用接口
  hasher hash_function() const;
  key_equal key_eq() const;
  void rehash(const bucket_traits & new_bucket_traits) ;

  //克隆
  void clone_from(const set &, Cloner, Disposer) ;

  //使用处置器的操作
```

① unordered_set 有一个特殊的局部迭代器概念（local_iterator），本书并未涉及，读者可自行研究。

```
    iterator erase_and_dispose(iterator, Disposer) ;
    iterator erase_and_dispose(iterator, iterator, Disposer) ;
    size_type erase_and_dispose(const_reference, Disposer) ;
    void clear_and_dispose(Disposer) ;

    //使用键操作
    size_type count(const KeyType &, KeyHasher, KeyValueEqual);
    size_type erase(const KeyType &, KeyHasher, KeyValueEqual) ;
    size_type erase_and_dispose(const KeyType &, KeyHasher,
   KeyValueEqual, Disposer) ;
    iterator find(const KeyType &, KeyHasher, KeyValueEqual) ;
    std::pair< iterator, iterator >
      equal_range(const KeyType &, KeyHasher, KeyValueEqual) ;

    //带检查的插入操作
    std::pair< iterator, bool >
    insert_check(const KeyType &, KeyHasher ,
             KeyValueEqual, insert_commit_data &);
    iterator insert_commit(reference, const insert_commit_data &) ;

    //从值获得迭代器
    iterator iterator_to(reference);
  };
```

unordered_set 可以使用多达 10 个的元数据选项，常用的有 base_hook<>/member_hook<>/value_traits<>、constant_time_size<>、size_type<>、hash<>和 equal<>，前三个我们都已经很熟悉了，后两个的含义如下：

- hash<>　：散列容器使用的散列函数对象，缺省是 boost::hash<T>;
- equal<>：散列容器使用的相等比较函数对象，缺省是 std::equal_to<T>。

工厂元函数 make_unordered_set<>用于生产 unordered_set 容器。

6.6.5　unordered_set 的基本用法

unordered_set 属于半侵入式容器，故它的用法与纯侵入式容器有所不同——需要外部额外分配一个辅助内存空间来维护散列容器所需的负载因子。

辅助内存空间是一个 unordered_set::bucket_type 类型的桶数组（静态或动态均

可），每个无序容器必须使用专用的桶数组，其生命周期必须长于 unordered_set，也就是说必须在 unordered_set 销毁以后才能销毁，否则会发生未定义错误。

unordered_set 的构造函数需使用 unordered_set::bucket_traits 包装 bucket_type 数组作为参数初始化，例如：

```
typedef make_unordered_set<point>::type set_t;          //无序侵入式容器
set_t::bucket_type buckets[20];                         //辅助桶数组
set_t s(set_t::bucket_traits(buckets, 20));             //构造一个空容器
```

桶数组也可以使用动态数组：

```
std::vector<set_t::bucket_type> buckets(20);
set_t s(set_t::bucket_traits(&buckets[0], 20));
```

桶数组一旦被指定就不能变动，可以使用 rehash() 传入一个新的桶数组（原桶数组亦可）重散列。

unordered_set 的用法基本类似 set，但因为它是无序容器，故不要求元素有顺序关系（operator<），只要可以被散列和判等即可。

为了让 unordered_set 容纳 point 类，需要做如下的修改，增加相等比较和散列值计算：

```
class point: public unordered_set_base_hook<>              //基类挂钩
{
public:
    ...//数据成员和构造函数
    friend bool operator==(const point& l, const point& r) //相等操作
    {   return l.x == r.x && l.y == r.y;    }

    friend std::size_t hash_value(const point& p)          //散列函数，参见 4.1 节
    {
        size_t seed = 2011;
        hash_combine(seed, p.x);
        hash_combine(seed, p.y);
        return seed;
    }
};
```

示范 unordered_set 基本用法的代码如下：

```
int main()
{
  using namespace boost::assign;                  //assign 名字空间
  ptr_vector<point> vec =                         //指针容器
      ptr_list_of<point>(0)(1)(2)(3)(4);          //使用 assign 库初始化

  typedef make_unordered_set<point>::type set_t; //无序侵入式集合容器

  std::vector<set_t::bucket_type> buckets(2);     //定义一个很小的辅助空间
  set_t s(vec.begin(), vec.end(),                 //迭代器区间构造
      set_t::bucket_traits(&buckets[0], 2));      //指定桶

  assert(s.size() == 5);                          //获得容器大小
  assert(s.count(point(1)) == 1);                 //计算元素数量

  BOOST_FOREACH(point& p, s)                //无序容器只能正向遍历，无逆序遍历
  {
      cout << p.x << ",";                   //输出结果是无序的
  }

  s.erase(s.find(point(1)));                //查找操作
  assert(s.size() == 4);

  set_t::bucket_type buckets2[10];                    //定义一个新的辅助空间
  s.rehash(set_t::bucket_traits(buckets2, 10));       //重新散列

  s.clear();                                          //清空容器
};
```

6.6.6 unordered_set 的特有用法

unordered_set 具有与 set 类似的一些侵入式容器特有接口，包括克隆、处置器、迭代器特殊操作等，但稍有不同，区别如下：

- 没有 container_from_end_iterator()和 container_from_iterator()；
- 没有静态成员函数 s_iterator_to()；
- 不能执行容器比较操作。

示范这些特有接口的代码如下：

```
std::vector<point*> vec;                        //一个容纳原始指针的标准容器
for (int i = 0;i < 5; ++i)                      //添加 5 个指针
{
    vec.push_back(new point(i, i));
}

typedef make_unordered_set<point>::type set_t;

std::vector<set_t::bucket_type> buckets(5);     //辅助空间
set_t s(make_indirect_iterator(vec.begin()),    //迭代器区间构造
    make_indirect_iterator(vec.end()),          //使用间接迭代器
    set_t::bucket_traits(&buckets[0], 5));

//删除容器中的前两个元素
s.erase_and_dispose(s.begin(), boost::next(s.begin(), 2),
    disposer());

set_t::bucket_type buckets2[10];                    //定义一个新的辅助空间
set_t s2(set_t::bucket_traits(buckets2, 10));       //另一个无序容器

s2.clone_from(s, cloner(), disposer());             //克隆容器
assert(*s2.begin() == *s.begin());
```

使用键操作值

unordered_set 也可以使用键来操作值，但因为它是无序的，所以不使用比较函数对象 KeyValueCompare，而是使用 KeyHasher 和 KeyValueEqual，这两个函数对象分别对应于 equal<>和 hash<>，差别仅在于是对键操作，结果应与对值的操作一致。

point 的两个键操作函数对象可实现如下：

```
typedef std::pair<int, int> ukey_type;              //使用一个 pair 作为键
struct key_hasher                                   //散列函数对象
{
  std::size_t operator()(const ukey_type& k)        //算法应与 point 一致
  {
      size_t seed = 2011;
      hash_combine(seed, k.first);
      hash_combine(seed, k.second);
      return seed;
  }
```

```
};
struct key_equal                                              //相等函数对象
{
  bool operator()(const ukey_type& k, const point& p)
  {
      return k.first == p.x && k.second == p.y;
  }
  bool operator()(const point& p, const ukey_type& k )
  {
      return operator()(k, p);
  }
};
```

示范 unordered_set 键操作的代码如下：

```
int main()
{
  using namespace boost::assign;                         //assign 名字空间
  ptr_vector<point> vec =                                //指针容器
      ptr_list_of<point>(0)(1)(2)(3)(4);                 //使用 assign 库初始化

  typedef make_unordered_set<point>::type set_t;

  std::vector<set_t::bucket_type> buckets(5);            //辅助空间
  set_t s(vec.begin(), vec.end(),                        //区间构造
      set_t::bucket_traits(&buckets[0], 2));

  key_hasher kh;                                         //散列函数对象
  key_equal keq;                                         //相等函数对象

  assert(s.count(make_pair(1,0), kh, keq) == 1);         //计算元素数量
  assert(s.find(make_pair(2, 0), kh, keq) != s.end());   //查找元素

  s.erase(make_pair(4, 0), kh, keq);                     //删除元素
  assert(s.find(make_pair(4, 0), kh, keq) == s.end());

  point tmp(5, 5);                                       //一个新元素
  set_t::insert_commit_data idata;                       //用于提交的数据类型

  //带检查的插入操作
  assert(s.insert_check(make_pair(5, 5), kh, keq, idata).second);
  s.insert_commit(tmp, idata);
  assert(s.find(make_pair(5, 5), kh, keq) != s.end());
```

```
}
```

6.6.7 unordered_multiset 类摘要

unordered_multiset 与 std::multiset 类似，可以容纳任意的元素，允许重复，它的类摘要如下：

```
template<class T, class O1, ..., class O10>
class unordered_multiset
   : public make_unordered_multiset<T,...>::type
{...};//基本同 unordered_set
```

unordered_multiset 的接口与 unordered_set 基本相同，但因为它允许插入重复的元素，所以没有 insert_check()和 insert_commit()系列函数。

工厂元函数 make_unordered_multiset<>用于生产 unordered_multiset 容器。

6.6.8 unordered_multiset 的用法

unordered_multiset 的用法与 unordered_set 几乎是一样的，它们的区别仅在于是否允许重复的元素。

简单示范 unordered_multiset 用法的代码如下：

```
int main()
{
  using namespace boost::assign;                        //assign 名字空间
  ptr_vector<point> vec =                               //指针容器
      ptr_list_of<point>(0)(1)(2)(3)(4)(3)(2);          //使用 assign 库初始化

  typedef make_unordered_multiset<point>::type set_t;   //无序多键容器

  std::vector<set_t::bucket_type> buckets(5);
  set_t s(set_t::bucket_traits(&buckets[0], 2));        //一个空容器

  s.insert(vec.begin(), vec.end());                     //区间插入
  assert(s.size() == vec.size());

  assert(s.count(point(2)) == 2);                       //有重复元素
```

```
    //foreach 使用基于键的 equal_range()
    BOOST_FOREACH(point& p,
        s.equal_range(make_pair(3, 0), key_hasher(), key_equal()))
    {
        cout << p.x << ",";                              //输出两个元素
    }
}
```

6.7 其他议题

本节我们将讨论关于侵入式容器的一些其他议题。

6.7.1 同时使用多个挂钩

经过前几节的学习，我们已经了解了侵入式容器的用法，应该注意到侵入式容器的一个特点：它并不拷贝对象，也不分配内存，仅仅是把已有的对象链接在一起，因此，如果对象含有多个挂钩，那么就可以同时被多个（挂钩对应的）侵入式容器容纳，相当于给这些对象建立了多个不同索引方式的视图。

使用多个挂钩与使用单个挂钩没有什么区别，基类挂钩和成员挂钩可以任意混用，只是多个基类挂钩需要使用 tag<>选项区分。

下面我们修改 point 的定义，为它增加多个挂钩：

```
class point:
  public list_base_hook<>,                              //链表基类挂钩，缺省标签
  public unordered_set_base_hook<tag<struct us_tag>> //无序集合基类挂钩
{
public:
  ...//同前

  set_member_hook<> m_shook;                         //用于有序单键集合的成员挂钩
  set_member_hook<> m_mshook;                        //用于有序多键集合的成员挂钩

  friend bool operator==(const point& l, const point& r)
  {...}
  friend bool operator<(const point& l, const point& r)
  {...}
```

```
    friend std::size_t hash_value(const point& p)
    {...}
};
```

这里我们为 point 增加了四个挂钩，这意味着一个对象可以同时被放入四种不同的侵入式容器，示范代码如下：

```
int main()
{
    using namespace boost::assign;                  //assign 名字空间
    ptr_vector<point> vec =                         //指针容器
        ptr_list_of<point>(0)(1)(2)(3)(4)(3)(2);    //使用 assign 库初始化

    //链表容器
    typedef make_list<point>::type list_t;
    //无序集合容器
    typedef make_unordered_set<point,
        base_hook<unordered_set_base_hook<tag<us_tag>>> >::type uset_t;
    //有序单键集合容器
    typedef make_set<point,
        member_hook<point, set_member_hook<>, &point::m_shook>>::type set_t;
    //有序多键集合容器
    typedef make_multiset<point,
        member_hook<point, set_member_hook<>, &point::m_mshook>>::type mset_t;

    list_t lt(vec.rbegin(), vec.rend());            //逆序插入链表
    BOOST_FOREACH(point& p, lt)
    {
        cout << p.x << ",";
    }

    set_t s(lt.begin(), lt.end());                  //有序单键集合容器
    mset_t ms(lt.begin(), lt.end());                //有序多键集合容器
    assert(s.size() == 5 && ms.size() == 7);

    uset_t::bucket_type buckets[20];
    uset_t us(uset_t::bucket_traits(buckets, 20)); //无序集合容器

    us.insert(vec.begin(), vec.end());              //插入元素
    assert(us.size() == 5);

    lt.pop_front();                                 //链表弹出一个元素
```

```
    assert(lt.size() == 6 &&                          //链表容量减少
        s.size() == 5 && ms.size() == 7);             //其他容器不受影响

    set_t::iterator iter =
        set_t::s_iterator_to(*us.begin());            //从其他容器的迭代器获得迭代器
    cout << iter->x << endl;
};
```

6.7.2　链接模式

挂钩的链接模式（link mode）是一个非常重要的属性，因此我们有必要在这里再讨论一下。

链接模式有三种策略：safe_link、auto_unlink 和 normal_link。

safe_link 是最常用的一种链接模式，正如它的名字，让挂钩可以安全地处理链接。挂钩（也就是含有挂钩的节点类）构造时是未连接状态，挂钩析构时（即节点类被销毁）检查挂钩的连接状态，如果已连接则发生断言异常，从而保证了侵入式容器不会发生访问无效指针的错误。在插入容器时 safe_link 的挂钩也会检查连接状态，不允许重复插入同一个容器，移出容器时会自动把节点置为未连接状态，因此使用 safe_link 挂钩的容器总是安全的。

auto_unlink 挂钩的大部分行为与 safe_link 挂钩相同，但在析构时会自动调用算法的 unlink() 函数从容器中自我移除。auto_unlink 挂钩同时还为节点提供了可用的 unlink() 成员函数，用户可以在容器之外任意地把节点移出侵入式容器。auto_unlink 的这些特点令它在某些时候很方便，但它的安全性要差一些，因为有可能容器内容在未经过容器操作的情况下就发生了改变，而且多线程时没有安全保证。还有一点，auto_unlink 不能使用常量时间的获取大小操作，也就是说必须在容器的模板参数中明确配置 constant_time_size<false>。

最后的 normal_link 是一个完全无任何操作的“空模式”，节点的构造、析构、插入、移除时都不做任何检查，适用于我们需要完全手工管理节点类的情形。

本书中我们主要使用的是 safe_link 模式，另外两种模式则很少使用，感兴趣的读者可以自行实践它们的用法。

6.7.3　万能挂钩

同时使用多个挂钩（6.7.1 小节）是一个很有用的功能，但在被侵入类编写不同容器对应的挂钩代码显得有些麻烦，intrusive 库为此在头文件<boost/intrusive/any_hook.

hpp>特别提供了可用于任意侵入式容器的“万能”式挂钩——any_base_hook 和 any_member_hook。

any_base_hook 和 any_member_hook 可以替代任意的基类挂钩和成员挂钩，选项参数和用法完全相同，但不能使用 auto_unlink 连接模式，例如：

```
class point:
  public any_base_hook<>
{
public:
  ...//数据成员，同前
  any_member_hook<> m_shook;
  any_member_hook<> m_mshook;
};
```

any_base_hook 和 any_member_hook 在用于配置容器时需要使用辅助元数据 any_to_xxx_hook<>，它用来把 any_hook 转换成特定容器所需的挂钩选项，代码如下所示：

```
typedef any_to_list_hook<                       //转换为链表挂钩
    base_hook<any_base_hook<>>> list_hook_opt;
typedef make_list<point, list_hook_opt>::type list_t;

typedef any_to_set_hook<member_hook<point,      //转换为有序单键集合挂钩
    any_member_hook<>, &point::m_shook>> set_hook_opt;
typedef make_set<point, set_hook_opt>::type set_t;

typedef any_to_set_hook<member_hook<point,      //转换为有序多键集合挂钩
    any_member_hook<>, &point::m_mshook>> mset_hook_opt;
typedef make_multiset<point, mset_hook_opt>::type mset_t;
```

6.8 总结

本章我们研究了侵入式容器库 intrusive，它同指针容器一样，也是一个比较庞大的库，内容很丰富。

侵入式容器历史悠久，但大多数都需要由程序员自己手工编写，自 STL 出现以后用者寥寥，boost.intrusive 则构建了全新的侵入式容器框架，提供了可与 STL 媲美的大量有用的侵入式容器，而且具有标准容器所缺乏的一些特性。

侵入式容器本质上属于链式容器，使用指针连接各个节点，本身不具有内存管理功能，

所以元素的生命周期管理是一个关键点，在使用侵入式容器时必须保证所有元素的生命周期都要长于侵入式容器，否则就可能会发生未知错误。如果要动态创建侵入式容器的元素，我们最好使用指针容器来简化内存管理工作。

侵入式容器定义了节点、算法、挂钩、选项、处置器和克隆等一系列的基本概念，用到了大量的模板元编程技术，这些概念中最重要的是挂钩和选项。挂钩可分为基类挂钩和成员挂钩，两者的用法不同但效果相同，适用于不同需求的场合。选项（options）是元编程的一个很好的具体应用例子，它可以无视顺序对元数据打包，在编译器配置侵入式容器的各个特性，提高了侵入式容器的运行时效率。由于元编程使用的类型声明比较复杂，通常需要适时使用 typedef 来使代码更加清晰易读。

intrusive 库提供了序列、有序集合和无序集合三大类侵入式容器，它们基本与标准容器等价，接口也非常类似，因而学习成本较低，易于掌握。但侵入式容器也有一些不同于标准容器的特殊操作，如处置器、克隆器、键-值操作等，这些操作的工作原理略为复杂，需要对侵入式容器的实现机制有较深刻的了解才能较好地掌握。

关于侵入式容器还有时间复杂度、空间复杂度的定性定量分析，以及定制值特征等更高级的内容，限于作者水平暂不做讨论，请读者见谅。

第7章

多索引容器

在前两章里我们已经看到了Boost提供的指针容器和侵入式容器，本章我们来研究本书中的最后一种特殊容器——multi_index_container（多索引容器），它有些类似标准容器，但可以同时提供多种对元素的访问方式——很像是数据库领域中的多索引机制。

读者可以把多索引容器与前两章的指针容器和侵入式容器对比学习，三者之间有许多相似之处，通过交叉参考能够更好地理解这三大容器库。

7.1 概述

C++98 标准库提供的容器（如list、set）能够容纳可拷贝可赋值的元素，它们对外部呈现的访问接口反映了内部的存储结构，只能以一种已经确定的顺序访问元素（例如list就只能顺序访问元素）。由于STL的标准性和权威性，这个设计准则也被许多其他仿STL的容器所接受，比如boost.unordered。

但有的时候这种固定顺序的访问方式会带来不便，我们可能会想对同一组元素执行不同的访问顺序准则，比如即要以顺序方式遍历list里的元素，又想以大小排序的方式遍历元素，这时使用STL容器显然不能满足要求。

指针容器（第 5 章）和侵入式容器（第 6 章）为这个问题提供了一定程度上的解决方案。指针容器可以使用视图分配器（view_clone_allocator）对另一个指针容器建立视图，从而在不改变原容器的情况下提供新的访问方式；侵入式容器直接修改元素的结构，可以添加任意多个挂钩，每一个挂钩都可以对应一种访问接口，但这两种解决方案也有局限性：指针容器要求元素必须是专有（exclusive）的，有时还要实现克隆概念；而侵入式容器则必

须修改元素的定义，而很多时候这是不被允许的。

boost.multi_index库是对这一问题的完美解决方案。从名字上就可以知道，它提供了“多索引”的容器，能够以不同的索引方式访问同一组存储在容器中的元素，并且没有指针容器或者侵入式容器的特殊要求。

multi_index位于名字空间boost::multi_index，为了使用multi_index组件，需要包含头文件<boost/multi_index_container.hpp>和其他的一些索引信息头文件，即：

```
#include <boost/multi_index_container.hpp>    //容器头文件
#include <boost/multi_index/xxx.hpp>          //索引头文件如 ordered_index.hpp
using namespace boost::multi_index;           //打开名字空间
```

本书有时候为了特别强调也会使用名字空间别名mi：

```
namespace mi = boost::multi_index;
```

7.2 入门示例

同前两章的指针容器和侵入式容器一样，对于multi_index这种新式容器我们也用一些简单的代码来演示多索引容器的基本用法和功能，帮助读者快速了解multi_index库。

7.2.1 简单的例子

第一个例子非常简单，它虽然使用了多索引容器，但仅有一个索引，用法与标准容器set无异：

```
#include <boost/assign.hpp>
#include <boost/foreach.hpp>
#include <boost/multi_index_container.hpp>          //多索引容器头文件
#include <boost/multi_index/ordered_index.hpp>      //有序索引
using namespace boost::multi_index;                 //打开名字空间

int main()
{
  multi_index_container<int> mic;           //一个多索引容器，缺省使用有序索引
  assert(mic.empty());                      //缺省容器为空

  mic.insert(1);                            //插入一个元素
```

```
    assert(mic.size() == 1);                    //使用 size()成员函数查看元素数量

    using namespace boost::assign;              //打开 assign 库名字空间
    insert(mic)(2), 7, 6, 8;                    //使用 assign 库的插入函数添加元素

    assert(mic.size()  == 5);
    assert(mic.count(2) == 1);                  //使用 count()函数计算元素的数量
    assert(mic.find(10) == mic.end());          //可以查找元素

    BOOST_FOREACH(int i, mic)                   //可以使用 foreach 算法
    {
        cout << i << ',';                       //元素顺序输出：1，2，6，7，8
    }
}
```

这段代码非常短小，除了容器的声明是multi_index_container<int>，其他部分与set<int>没有任何差异，从中我们可以看到多索引容器的几个基本特性，如下：

- 多索引容器可以提供一个或多个索引接口，也就是说不一定是“多索引”；
- 多索引容器缺省使用有序索引，类似 std::set，是一个有序单键集合；
- 多索引容器提供与标准容器用法完全一致的接口，如 insert()、find()等成员函数，因而很容易学习和使用；
- 多索引容器也能够使用标准容器的辅助工具，如 Boost 的 foreach 算法和 assign 库，大大增强了它的实用价值。

当然，只有一个索引的多索引容器的意义是不大的，这实际上是多索引容器的退化形式，接下来我们看第二个例子，它是真正的“多索引容器”。

7.2.2　复杂的例子

第二段示例代码略微有些复杂，其重点在多索引容器的类型定义部分，它实质上是一个模板元编程领域中的元函数概念：

```
#include <boost/assign.hpp>
#include <boost/foreach.hpp>
#include <boost/multi_index_container.hpp>          //多索引容器头文件
#include <boost/multi_index/ordered_index.hpp>      //有序索引
#include <boost/multi_index/hashed_index.hpp>       //散列(无序)索引
```

```
#include <boost/multi_index/key_extractors.hpp>      //键提取器
using namespace boost::multi_index;
namespace mi = boost::multi_index;

int main()
{
  typedef multi_index_container<int,         //多索引容器，元素类型为 int
      indexed_by<                            //使用 index_by 元函数定义多个索引
          ordered_unique<mi::identity<int>>, //第一个是有序单键索引
          hashed_unique<mi::identity<int>>   //第二个是散列（无序）单键索引
          >                                  //索引定义结束
      > mic_t;                               //多索引容器定义结束
  mic_t mic;                                 //一个多索引容器，有两个索引

  using namespace boost::assign;             //同样可以使用 assign 库
  insert(mic)(2), 1, 7, 6, 8;                //默认使用第一个索引操作，即有序索引

  assert(mic.size() == 5);                   //默认第一个索引的接口同 std::set
  assert(mic.count(2) == 1);
  assert(mic.find(10) == mic.end());

  //使用模板成员函数 get<>()获得第二个索引，编号从 0 算起
  mic_t::nth_index<1>::type& hash_index = mic.get<1>();

  assert(hash_index.size() == 5);            //第二个索引的用法同 unordered 容器
  assert(hash_index.count(2) == 1);
  assert(hash_index.find(10) == hash_index.end());

  BOOST_FOREACH(int i, hash_index)           //同样可以使用 foreach 算法
  {
      cout << i << ",";
  }
}
```

这段代码我们需要注意的有下面几个地方：包含的头文件、多索引容器的定义和第一个索引以外的其他索引的获取与使用方法。

要使用multi_index库提供的其他索引方式必须在<boost/multi_index_container.hpp>之外手工添加所需的索引头文件，例如有序索引需使用<boost/multi_index/ordered_index.hpp>，散列（无序）容器需使用

<boost/multi_index/hashed_index.hpp>，为了定义键的索引方式还需要包含键提取器头文件<boost/multi_index/key_extractors.hpp>。

接下来的重点是多索引容器的定义。多索引容器的类型是multi_index_container，它有两个模板参数。第一个是容纳的元素类型，对元素的要求与标准容器略低，必须是可拷贝的（不必是可赋值）。第二个模板参数用来声明容器上的索引类型，是一个indexed_by<>结构，它可以接受数个不同的索引，在本例中使用了一个有序单键索引（ordered_unique<>）和一个无序单键索引（hashed_unique<>）。索引又是一个元数据，它需要使用一种被称为键提取器（key extractor）的函数对象来获得用于产生索引的键类型，在这里我们直接使用元素本身作为键，所以使用mi::identity<>[①]。这样我们就成功地定义了一个具有两个索引的多索引容器。由于多索引容器的定义比较复杂，通常我们需要使用typedef来简化类型定义。

如果不显示指定索引，容器将使用第一个索引，效果和用法同之前的例子。想要获得第一个以外的索引就要使用模板成员函数get<N>()，它返回一个特定的索引类型的引用："mic_t::nth_index<N>::type&"。如果不关心具体的类型信息我们也可以直接使用boost.typeof来简化变量的声明（注意必须是引用）：

```
BOOST_AUTO(&hash_index , mic.get<1>());     //正确，使用&声明引用变量
BOOST_AUTO(hash_index , mic.get<1>());      //错误，不能声明索引实例变量
```

索引就是多索引容器的访问接口，每个索引的用法都如同标准容器一样，只是访问准则是在最开始声明多索引容器时就已经确定好的。在这段代码中第一个索引是有序单键索引，相当于std::set，第二个索引是散列（无序）单键索引，相当于boost::unordered。除了类型不同，这些索引的用法并无太多特殊之处。

7.2.3 更复杂的例子

最后这个例子中我们将一个具有较复杂结构的自定义类：person，作为我们后续讨论的元素类型，它实现了<、==等比较操作和散列运算，定义如下：

```
class person:
  boost::less_than_comparable<person>     //使用 operators 库实现全序比较
{
public:
  int m_id;                               //身份标识号
```

① 使用名字空间 mi 限定 identity<>的原因是在 STLport 的实现中有一个同名的函数对象，如果没有名字空间限定会发生编译错误。当然，如果不使用 STLport 就可以不用这么写。

```
  string m_fname, m_lname;                //姓名

  person(int id, const string& f, const string& l):       //构造函数
      m_id(id), m_fname(f), m_lname(l){}

  const string& first_name() const        //const 成员函数，取 m_fname
  {   return m_fname;}

  string& last_name()                     //非 const 成员函数，取 m_lname
  {   return m_lname;}

  friend bool operator<(const person&l, const person& r)     //比较操作
  {   return l.m_id < r.m_id;}

  friend bool operator==(const person& l, const person& r)   //相等操作
  {   return l.m_id == r.m_id ;}

  friend std::size_t hash_value(const person& p)     //散列函数，参见 4.1 小节
  {
      size_t seed = 2011;
      hash_combine(seed, p.m_fname);
      hash_combine(seed, p.m_lname);
      return seed;
  }
};
```

下面的代码定义了一个有四个索引的容器：

```
#include <boost/assign.hpp>
#include <boost/typeof/typeof.hpp>
#include <boost/foreach.hpp>
#include <boost/multi_index_container.hpp>             //多索引容器头文件
#include <boost/multi_index/ordered_index.hpp>         //有序索引
#include <boost/multi_index/hashed_index.hpp>          //散列(无序)索引
#include <boost/multi_index/sequenced_index.hpp>       //序列索引
#include <boost/multi_index/key_extractors.hpp>        //键提取器
using namespace boost::multi_index;

int main()
{
```

```
typedef multi_index_container<                    //多索引容器
    person,                                       //容器的元素类型为 person
    indexed_by<                                   //使用 index_by 元函数定义多个索引
        sequenced<>,                              //第一个是序列索引
        ordered_unique<identity<person>>,         //第二个是有序单键索引
        ordered_non_unique<                       //第三个是有序多键索引
            member<person, string, &person::m_fname>>, //使用成员变量
        hashed_unique<identity<person>>           //第四个是散列（无序）单键索引
    >                                             //索引定义结束
> mic_t;                                          //多索引容器定义结束

mic_t mic;                                        //声明多索引容器变量

using namespace boost::assign;                    //使用 assign 库
push_front(mic) //第一个索引是序列索引，所以可以使用 push_front()
    (person(2, "agent", "smith"))                 //插入 4 个元素
    (person(20, "lee", "someone"))                //顺序无关紧要
    (person(1, "anderson", "neo"))                //id 不可重复
    (person(10, "lee", "bruce"));                 //m_fname 可重复

BOOST_AUTO(&index0, mic.get<0>());                //使用 typeof 获得四个索引
BOOST_AUTO(&index1, mic.get<1>());
BOOST_AUTO(&index2, mic.get<2>());
BOOST_AUTO(&index3, mic.get<3>());

//使用索引 0，顺序输出元素，没有排序
BOOST_FOREACH(const person& p, index0)
{
    cout << p.m_id << p.m_fname << ",";
}

//索引 1，获得 id 最小的元素
assert(index1.begin()->m_id == 1);

//索引 2，允许重复键的元素
assert(index2.count("lee") == 2);

//使用 assign::insert 在索引 2 上插入重复键的元素
insert(index2)(person(30, "lee", "test"));
assert(index3.size() == 5);
```

```
    //插入 id 重复的元素因索引 1 的限制而不能成功
    assert(!index2.insert(person(2, "lee", "test2")).second);
}
```

上面这段代码中的多索引容器明显比前两个例子要复杂很多，它一共定义了四个索引，我们可以用序列方式、有序单键方式、有序多键（基于m_fname）和无序单键方式来访问同一组元素。因为多索引容器的定义很复杂，所以良好的缩进格式和注释是很有必要的。

容器的第一个索引是sequenced<>，它并不对元素排序，所以通常不需要模板参数；第二个索引是ordered_unique<>，它是有序单键索引，要求元素具有operator<；第四个索引是hashed_unique<>，它是无序单键索引，要求元素满足boost.hash可以计算散列值；第三个索引是ordered_non_unique<>，它使用了一个不同于identity的新的键提取器member，形式上类似侵入式容器的member_hook<>（参见 6.4.4 小节），可以使用元素的成员变量作为键。

这些索引都可以使用模板成员函数get<N>()获得，彼此的读操作是独立的，但插入删除等变动元素的操作可能会受到其他索引的制约。比如代码的最后三行，第三个索引可以向容器里插入任意m_fname重复的元素，但不能插入m_id重复的元素，因为第二个索引被声明为ordered_unique<>，它的operator<()使用了m_id成员，所以必须是唯一的。

7.3 基本概念

通过前面的三个例子我们已经对多索引容器有了初步的了解，本节将介绍multi_index库中构建多索引容器的基本概念，这些概念的实现大量使用了模板元编程，本书节选代码时从略。

7.3.1 索引

索引（index）是multi_index库中最重要的概念，它是多索引容器里访问元素的接口，决定了外部用户以何种准则对元素执行读/写/查找等操作。不同的索引有不同的使用接口，索引通常的形式如下：

```
template<typename KeyFromValue,...>
class some_index
{
   public:
   typedef some_define key_type;
```

```
    typedef some_define value_type;
    ... //其他类型定义

   iterator begin();
   iterator end();
   bool      empty();
   size_type size();
    ... //其他成员函数
};
```

索引的接口定义完全模仿了标准容器，具有大多数常用的成员函数，所以在使用索引时完全可以把它当做是标准容器，但索引也有不同于标准容器的地方：索引通常不允许直接修改元素以保障索引的结构不会被意外破坏，而且因为一个多索引容器可能持有多个索引，故操作元素时可能存在相互的制约关系，某些标准容器的操作可能会因此无法执行。

虽然索引是确定的类型，但它被multi_index库定义为一个内部使用的类型（在boost::multi_index::detail子名字空间里），不能单独使用，也就是说我们无法自行声明一个索引类型的变量。索引的使用必须依附于多索引容器，在multi_index_container的模板参数中用索引说明（index specifier）定义，然后用模板成员函数get<>()来获得引用才能操作。

7.3.2 索引说明

索引的定义需要在multi_index_container的模板参数中使用索引说明（index specifier）。索引说明是一个高阶元数据，它使用键提取器和其他参数来定义索引，内部有两个名为node_class<>和index_class<>的元函数，返回可供容器使用的节点类型和索引类型。

索引说明的基本形式如下：

```
template<typename Arg1,typename Arg2,...>
struct some_index_specifier
{
  template<typename Super>
  struct node_class
  {
    typedef some_define type;
  };

  template<typename SuperMeta>
```

```
    struct index_class
    {
      typedef some_define type;
    };
};
```

目前multi_index库提供以下四类索引说明（为叙述方便，之后有时会将索引说明简称为索引，请读者注意）：

- 序列索引 ：这类索引只有一个 sequenced，是类似 std::list 的双向链表序列访问接口，见 7.5 小节；
- 随机访问索引 ：这类索引只有一个 random_access，是类似 std::vector 的序列访问接口，提供 operator[]方式的随机访问能力，见 7.6 小节；
- 有序索引 ：包括 ordered_unique 和 ordered_non_unique，是类似 std::set 和 std::multiset 的有序集合访问接口，见 7.7 小节；
- 散列（无序）索引：包括 hashed_unique 和 hashed_non_unique，是类似 boost::unordered_set 和 boost::unordered_multiset 的无序集合访问接口，见 7.8 小节。

7.3.3 键提取器

键提取器（key extractor）是一个单参函数对象，它可以从元素或元素的boost.ref包装中获得用做索引（排序）的键，通常被用作索引说明的第一个模板参数，其简要形式如下：

```
template<typename Arg1,typename Arg2,...>
struct some_key_extractor
{
  typedef some_define result_type;
  some_type& operator()(...)const;
};
```

multi_index库提供了以下六种键提取器，可以适用于各种情况：

- identity：使用元素本身作为键；
- member ：使用元素的某个 public 成员变量作为键；

- const_mem_fun：使用元素的某个 const 成员函数返回值作为键；
- mem_fun　　　：使用元素的某个非 const 成员函数返回值作为键；
- global_fun　　：使用操作元素的某个全局函数或静态成员函数的返回值作为键；
- composite_key：可以把以上的键提取器组合为一个新的键。

键提取器获取的键类型必须能够满足索引的要求，例如有序索引要求键定义了比较操作符，散列索引要求键可以计算散列值和执行相等比较。这些键提取器的具体讨论参见 7.4 小节。

7.3.4 索引说明列表

索引说明列表（index specifier list）是多索引容器multi_index_container的第二个模板参数，实现为一个indexed_by结构，用来定义多索引容器使用的索引，类摘要如下：

```
template<typename T0, typename T1, ...>
struct indexed_by: mpl::vector<T0, T1, ...>
{};
```

indexed_by基于模板元编程库mpl里的元数据序列mpl::vector（见 11.4.2 小节），是一个类型的容器，可以容纳多个索引说明。它使用了预处理元编程，当前的实现限定最多可容纳 20 个元数据，也就是说最多支持在一个容器上同时使用 20 个索引（相信已经足够用了）。

7.3.5 索引标签

多索引容器通常会持有多个索引，这些索引可以使用容器的模板成员函数get<N>()获取，其中的N是索引在索引说明列表中的序号（从 0 算起），但单纯使用序号来获取索引很不直观，不方便记忆，所以multi_index库提供索引标签（tag）的概念，允许使用语法标签来访问索引。

索引标签的定义使用模板类tag，它的类摘要如下：

```
template<typename T0, typename T1, ...>
struct tag
{
  typedef some_define type;
};
```

tag也是一个mpl类型容器，支持最多 20 个类型同时作为索引标签。标签类型是任意的，不仅可以使用自定义类型，甚至还可以使用C++内建类型如int、std::string。

使用索引标签需要把tag<>作为索引说明的第一个模板参数，键提取器需在标签之后，例如：

```
ordered_unique<tag<struct id>,...>    //使用自定义类型 struct id 作为标签
hashed_unique<tag<int,short>,...>     //使用内建整数类型 int、short 作为标签
```

但在同一个多索引容器中不允许使用重复的标签，否则会发生编译错误，例如：

```
ordered_unique<tag<struct id, id>,...>    //自定义标签重复
hashed_unique<tag<int,id>,...>            //与上一个索引的标签重复
```

模板成员函数get<>()有针对索引标签的重载形式，可以使用标签来获得索引，如下：

```
BOOST_AUTO(&ordered_index, mic.get<id>());
BOOST_AUTO(&hashed_index , mic.get<int>());
```

7.3.6 多索引容器

在multi_index库中多索引容器的类型是multi_index_container，它的声明如下：

```
template<
  typename Value,                                               //元素类型
  typename IndexSpecifierList=                                  //索引说明列表
             indexed_by<ordered_unique<identity<Value> > >,     //缺省值
  typename Allocator=std::allocator<Value> >                    //内存分配器
class multi_index_container;
```

multi_index_container的类声明可以与标准库的容器进行对比学习：模板参数分别是元素类型，排序准则和分配器类型，只不过排序准则非常复杂，是一个indexed_by类型的索引声明列表。索引说明列表缺省提供一个ordered_unique<>，意味着如果不指定索引说明，多索引容器的默认行为同std::set，是一个不允许重复的有序集合。

multi_index_container很像是一个索引的管理器，自身并没有太多的元素操作能力，大多数时候我们都通过get<>()以索引序号或者索引标签来获得索引再使用。

多个索引的好处不只是提供多个不同形式访问接口那么简单，我们还可以充分利用不同存储结构的时间优势，例如使用序列索引顺序存储元素，再用散列索引实现对元素的快速查找，用有序索引实现对元素的排序，多索引容器为我们提供了强大的数据操作能力。

7.4 键提取器

multi_index库强化了标准库中的键（key）概念，把原始的键类型（key type）扩展为一个功能更强的函数对象——键提取器，可以从元素中提取关联的用于排序的键，是multi_index库建立索引的基础。

键提取器分为可写和只读两类，可写键提取器总可以返回一个非常量的键引用，不仅可以读也可以写，而只读键提取器总返回常量的键引用，只能读。大多数索引只要求只读键提取器，仅在有序索引和散列索引使用modify_key()时才要求可写键提取器（参见 7.9.3 小节）。

7.4.1 定义

键提取器是一个函数对象，它的基本形式如下：

```
struct some_key_extractor
{
  typedef T result_type;                          //返回类型定义

  T& operator()(T& x)const;                       //操作原始类型
  T& operator()(const reference_wrapper<T>& x)const; //操作引用包装类型

  template<typename ChainedPtr>
  Type& operator()(const ChainedPtr& x)const;     //操作链式指针类型
};
```

键提取器的operator()不仅可以操作类型本身（T&），也可以操作被boost.ref库包装的对象（reference_wrapper<T>&），而且它还支持"链式指针"（chained pointer）对象。

所谓"链式指针"是指一系列类似指针对象的组合——包括原始指针、智能指针、迭代器等一切具有operator*可以执行解引用操作的类型，T*、T**、shared_ptr<T*>都属于链式指针类型，它们可以被连续递归解引用得到一个非指针类型T&或者reference_wrapper<T>&。键提取器的这个特性可以让我们轻松地操作指针，让多索引容器可以轻松地容纳指针类型的元素。

multi_index库所有预定义的键提取器均位于头文件<boost/multi_index/key_

extractors.hpp>，如果想要减少编译的时间（元计算的时间）也可以视需要只包含必要的头文件。

接下来我们逐个介绍这些键提取器，但由于composite_key用法比较复杂，将留在 7.11 小节学习。

7.4.2 identity

identity是一个最简单的键提取器，它不做任何“提取”动作，直接使用元素本身作为键，相当于标准容器的键类型（key type）。只要元素类型不是const，那么它就是可写的键提取器。

identity位于头文件<boost/multi_index/identity.hpp>，其类摘要如下：

```
template<class Type>
struct identity:
  mpl::if_c<
    is_const<Type>::value,
    detail::const_identity_base<Type>,
    detail::non_const_identity_base<Type>
  >::type
{};
```

identity使用了元函数if_c<>，根据类型Type是否被const修饰分别转交给const_identity_base和non_const_identity_base处理，这两个实现类的具体代码差异很小，下面列出const_identity_base的主要代码：

```
template<typename Type>
struct const_identity_base
{
  typedef Type result_type;          //返回类型定义

  //操作元素类型本身
  Type& operator()(Type& x)const
  {    return x; }

  //操作元素类型的 reference_wrapper 包装
  Type& operator()(const reference_wrapper<Type>& x)const
  {    return x.get(); }

  //操作链式指针
```

```
    template<typename ChainedPtr>
    typename disable_if<
            is_convertible<const ChainedPtr&,Type&>,Type&>::type
    operator()(const ChainedPtr& x)const
    {   return operator()(*x); }
};
```

const_identity_base的前两个operator()都很好理解，它们直接返回变量自身。最后一个重载形式用于处理链式指针，它使用了10.1小节介绍的元函数disable_if<>，当模板参数ChainedPtr是指针类型时编译器递归生成解引用的operator()，这样在运行时就可以连续调用直至获得最终的Type&类型。

离开多索引容器的范围，identity就是一个普通的函数对象，像是对类型做了一层薄薄的包装，例如：

```
assert((is_same<string, identity<string>::result_type>::value));
assert(identity<int>()(10) == 10);
assert(identity<string>()("abc") == "abc");

int *p = new int(100);                  //指针
int **pp  = &p;                         //指针的指针(链式指针)
assert(identity<int>()(pp) == 100);     //从链式指针中获得键
```

在多索引容器里identity可以用于简单的类型（如int、string）或者本身已经定义了比较、散列操作的类型，如之前的person。

7.4.3　member

键提取器member有些类似非标准函数对象select1st的功能，可以提取类型里的某个public成员变量作为键。它的接口、实现与identity基本相同，支持从类型本身、ref包装和链式指针里提取键。只要元素类型不是const，那么它就是可写的键提取器。

member位于头文件<boost/multi_index/member.hpp>，类摘要如下：

```
template<class Class,typename Type,Type Class::*PtrToMember>
struct member:
  mpl::if_c<
    is_const<Type>::value,
    detail::const_member_base<Class,Type,PtrToMember>,
    detail::non_const_member_base<Class,Type,PtrToMember>
  >::type
{};
```

member有三个模板参数，形式上很像侵入式容器的member_hook<>选项（参见 6.4.4 小节），指定要提取的类型Class、键类型Type（即类型的成员变量类型）以及成员变量指针PtrToMember，例如：

```
typedef pair<int, string> pair_t;
pair_t p(1, "one");

assert((member<pair_t, int, &pair_t::first>()(p) == 1));
assert((member<pair_t, string, &pair_t::second>()(p) == "one"));

person per(1, "anderson", "neo");
assert((member<person, int, &person::m_id>()(per) == 1));
```

某些对C++标准支持较差的编译器（如VC6）可能无法使用member，所以multi_index提供了一个等价的替代品：member_offset，它使用偏移量来代替成员指针，本书对它不做讨论，读者有需要可阅读Boost文档（用法很简单）。

为了获得最大的兼容性并且方便使用，multi_index库定义了一个宏BOOST_MULTI_INDEX_MEMBER，它无须我们手工写出成员变量指针的声明，而且会根据编译器的能力自动选用member或者member_offset：

```
#define BOOST_MULTI_INDEX_MEMBER(Class,Type,MemberName)\
```

BOOST_MULTI_INDEX_MEMBER有三个参数，前两个Class和Type与member的模板参数定义相同，后一个是成员变量名，不需要写出取地址操作符和类名限定。对于大多数支持成员指针模板参数的编译器，宏展开如下：

```
::boost::multi_index::member< Class,Type,&Class::MemberName >
```

使用宏BOOST_MULTI_INDEX_MEMBER可以改写上面的代码如下：

```
assert(BOOST_MULTI_INDEX_MEMBER(pair_t, int, first)()(p) == 1);
assert(BOOST_MULTI_INDEX_MEMBER(pair_t, string, second)()(p) == "one");
assert(BOOST_MULTI_INDEX_MEMBER(person, int, m_id)()(per) == 1);
```

显然，因为无须写出成员变量指针，所以宏的写法更简单清晰，只是宏的名字略长，算是个小缺点。

7.4.4 const_mem_fun

const_mem_fun使用类型中的某个const成员函数返回值作为键，有些类似函数对象

mem_fn（参见 4.2 小节），但它使用上有两个限制：只能调用const成员函数，而且这个成员函数必须是无参调用。

const_mem_fun是一个只读键提取器，位于头文件<boost/multi_index/mem_fun.hpp>，它的类摘要如下：

```
template<class Class,typename Type,
Type (Class::*PtrToMemberFunction)()const>
struct const_mem_fun
{
  typedef typename remove_reference<Type>::type result_type;

  Type operator()(const Class& x)const
  {
    return (x.*PtrToMemberFunction)();          //调用无参 const 成员函数
  }
  ...//其他 operator()定义
};
```

const_mem_fun的模板参数与member类似，但最后一个参数是成员函数指针，同时Class和Type参数必须与成员函数指针精确匹配，示范代码如下：

```
string str("abc");
typedef const_mem_fun<string, size_t, &string::size > cmf_t;
assert(cmf_t()(str) == 3);

person per(1, "anderson", "neo");
typedef const_mem_fun<person, const string&, &person::first_name> cmf_t2;
assert((cmf_t2()(per) == "anderson"));
```

下面的两行代码会因为类型错误而无法编译。

```
typedef const_mem_fun<const person, const string&, &person::first_name>
                        cmf_t2;
 typedef const_mem_fun<person, string&, &person::first_name> cmf_t2;
```

const_mem_fun也有一个用于屏蔽编译器差异的宏BOOST_MULTI_INDEX_CONST_MEM_FUN，定义如下：

```
#define BOOST_MULTI_INDEX_CONST_MEM_FUN(Class,Type,MemberFunName) \
::boost::multi_index::const_mem_fun<
                                Class,Type,&Class::MemberFunName >
```

宏BOOST_MULTI_INDEX_CONST_MEM_FUN可以这样使用：

```
typedef BOOST_MULTI_INDEX_CONST_MEM_FUN(string, size_t, size) cmf_t;
typedef BOOST_MULTI_INDEX_CONST_MEM_FUN(person, \
                            const string&, first_name) cmf_t2;
```

7.4.5 mem_fun

mem_fun与const_mem_fun类似，使用元素的某个成员函数返回值作为键，但它只适用于成员函数是非const的情形。

mem_fun是一个只读键提取器，位于头文件<boost/multi_index/mem_fun.hpp>，它的类摘要如下：

```
template<class Class,typename Type,
        Type (Class::*PtrToMemberFunction)() >
struct mem_fun
{
  typedef typename remove_reference<Type>::type result_type;
  ...// operator()定义，同 const_mem_fun
};
```

注意：mem_fun与标准库的函数对象std::mem_fun重名，因此使用时可能需要加名字空间boost::multi_index限定，例如：

```
person per(1, "anderson", "neo");
typedef mi::mem_fun<person, string&, &person::last_name> mf_t;
assert((mf_t()(per) == "neo"));
```

我们也可以使用宏BOOST_MULTI_INDEX_MEM_FUN，它的声明如下：

```
#define BOOST_MULTI_INDEX_MEM_FUN(Class,Type,MemberFunName) \
::boost::multi_index::mem_fun< Class,Type,&Class::MemberFunName >
```

宏BOOST_MULTI_INDEX_MEM_FUN不必考虑名字空间的问题（展开时已经有了名字空间限定），可以这样使用：

```
typedef BOOST_MULTI_INDEX_MEM_FUN(person, string&, last_name) mf_t;
```

我们在使用时必须注意mem_fun的特点：它操作的是非const成员函数，而索引的大多数函数的接口都是常量性的const T&，所以如果直接存储元素类型T会因为无法获取键而导致编译失败。不过如果容器存储的是指针T*，那么mem_fun就可以不受限制地使用。

7.4.6 global_fun

global_fun使用一个全局函数或静态成员函数来操作元素，函数的返回值作为键，它的实现类似identity，支持const或非const的函数。

global_fun是一个只读键提取器，位于头文件<boost/multi_index/global_fun.hpp>，它的类摘要如下：

```
template<class Value,typename Type,Type (*PtrToFunction)(Value)>
struct global_fun:
  mpl::if_c<...>::type
{};
```

global_fun的用法类似const_mem_fun和mem_fun，只是最后一个模板参数必须是一个参数为Value类型的函数指针。

我们为person类定义一个全局函数nameof()，它返回person的全名：

```
string nameof(const person& p)
{
   return p.m_fname + " "  +p.m_lname;
}
```

之后我们就可以使用global_fun：

```
person per(1, "anderson", "neo");
typedef global_fun<const person&, string, &nameof> gf_t;
assert(gf_t()(per) == "anderson neo");
```

使用global_fun同样要注意类型参数要与函数指针精确匹配，否则代码无法通过编译，例如：

```
typedef global_fun<person&, string, &nameof> gf_t;          //错误
typedef global_fun<const person&, string&, &nameof> gf_t;  //错误
```

7.4.7 自定义键提取器

键提取器实质上是一个单参函数对象，所以我们也可以不使用multi_index库预定义的键提取器，完全自行编写——只需要满足键提取器的定义即可（静态多态）。

自定义键提取器首先要满足标准函数对象的要求，定义有内部类型result_type，它同时也是键类型。然后要实现操作元素类型的operator()，必须是const成员函数，根据

需要可实现数个针对T&、reference_wrapper<T>&和ChainedPtr&的重载，但不必都实现。

例如，我们可以编写一个键提取器person_name，它实现与global_fun<const person&, string, &nameof>等价的功能：

```
struct person_name
{
  typedef string result_type;                          //返回值类型定义，必需
  result_type operator()(const person& p) const        //必须为 const
  {
      return p.m_fname + " "  +p.m_lname;
  }
  result_type operator()(person *const p) const        //支持容纳原始指针
  {
      return p->m_fname + " "  +p->m_lname;
  }
};
```

自定义键提取器的用法参见 7.10.2 小节。

7.5 序列索引

序列索引是multi_index库提供的最简单的一种索引，它实际上没有对元素做任何索引操作，仅仅是顺序存储元素。

序列索引位于头文件<boost/multi_index/sequenced_index.hpp>。

7.5.1 索引说明

序列索引的索引说明是sequenced，它的类摘要如下：

```
template <typename TagList = tag<> >
struct sequenced
{
  template<typename SuperMeta>
  struct index_class
  {
    typedef detail::sequenced_index<...> type;
  };
};
```

因为序列索引不基于键排序，所以sequenced不使用任何键提取器，只有一个TagList模板参数，用来给索引贴语法标签。

7.5.2 类摘要

序列索引使用的类是detail::sequenced_index，它提供类似std::list的双向链表操作，但有些接口是常量性的，不能随意修改元素。如果想要修改容器里的元素需要使用特别的成员函数，参见 7.9 小节。

sequenced_index的类摘要如下：

```
class sequenced_index
{
  public:
  typedef some_define value_type;
  typedef some_define iterator;
  typedef iterator const_iterator;
  ... //其他类型定义

  //赋值操作
  sequenced_index& operator=(const sequenced_index& x);
  void assign(InputIterator first,InputIterator last);
  void assign(size_type n,const value_type& value);

  //迭代器操作
  iterator begin();
  iterator end();
  iterator iterator_to(const value_type& x);

  //元素访问
  const_reference front()const;
  const_reference back()const;

  std::pair<iterator,bool> push_front(const value_type& x);
  void                     pop_front();
  std::pair<iterator,bool> push_back(const value_type& x);
  void                     pop_back();

  std::pair<iterator,bool> insert(iterator position,const value_type& x);
  void insert(iterator position,size_type n,const value_type& x);
```

```
  iterator erase(iterator position);
  iterator erase(iterator first,iterator last);
  ...//其他 remove()、unique()、splice()、sort()等操作

  ...//各种比较操作符定义
};
```

sequenced_index的接口与std::list基本相同，因而很容易使用，需要注意的有如下几点：

- 索引不提供 public 的构造函数和析构函数（由多索引容器处理），但可以使用 operator=和 assign()赋值，用法同标准容器；
- 解引用迭代器（begin()、rbegin()等）返回的都是 const 类型，所以不能使用迭代器修改元素；
- 访问元素的 front()和 back()函数返回的是 const 引用，不能修改元素；
- 因为可能存在其他索引的约束，push_front()、push_back()和 insert()可能会执行失败，所以它们的返回值是如同 set::insert()一样的一个 pair，second 成员表示操作是否成功；
- 与侵入式容器类似，索引提供 iterator_to()函数，可以从容器内一个元素的引用（不是拷贝）获得相应的迭代器。

7.5.3 用法

因为序列索引不使用键提取器，也不涉及元素的排序，因而用法非常简单，完全可以把它当做是std::list的一个等价物（但不允许直接修改元素的值）。

示范序列索引用法的代码如下：

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/sequenced_index.hpp>
using namespace boost::multi_index;            //打开名字空间

int main()
{
  typedef multi_index_container<int,           //存储 int 类型的元素
      indexed_by<sequenced<                    //使用序列索引
                 tag<int, struct int_seq> > >  //添加两个标签
      > mic_t;
```

```
    using namespace boost::assign;
    mic_t mic = (list_of(2),3,5,7,11);            //使用 assign 库直接初始化

    assert(!mic.empty() && mic.size() == 5);      //容器容量操作

    assert(mic.front() == 2);                 //容器缺省使用第一个索引，即序列索引
    assert(mic.back() == 11);

    assert(mic.push_front(2).second);             //目前没有其他索引约束
    assert(mic.push_back(19).second);             //所以总可以增加元素

    BOOST_AUTO(&seq_index, mic.get<int_seq>());   //使用标签获得索引

    seq_index.insert(seq_index.begin(), 5, 100);  //插入 5 个元素
    assert(std::count(seq_index.begin(), mic.end(), 100) == 5);

    seq_index.unique();                           //删除重复的元素
    assert(std::count(seq_index.begin(), mic.end(), 100) == 1);
}
```

序列索引也支持容器间的比较操作，与std::list一样：

```
mic_t mic1 = (list_of(2),3,5,7,11);     //初始化

mic_t mic2 = mic1;                      //赋值操作
assert(mic1 == mic2);                   //两个容器内的元素一致，比较相等

mic2.push_back(3);                      //添加一个元素
assert(mic1 < mic2);                    //比较不等
```

虽然序列索引很像std::list，但它与std::list的一个重要区别是不能随意修改元素的值，使用时要注意。例如下面的代码会发生编译错误：

```
*seq_index.begin() = 9; //编译错误，报 const 错
```

另外，成员函数iterator_to()的使用也值得注意，只有当参数确实是容器内元素的引用时才能返回正确的迭代器：

```
const int& x = mic.front();                   //获得元素的引用
assert(mic.begin() == mic.iterator_to(x));    //获取对应的迭代器
```

使用与元素等价的拷贝虽然也可以调用iterator_to()，但返回的迭代器与索引没有

任何关系，使用这个迭代器去执行其他操作会导致运行时错误：

```
assert(mic.begin() != mic.iterator_to(2));          //返回的迭代器不能使用
```

7.6 随机访问索引

随机访问索引是multi_index库提供的另一种序列索引，它同样是顺序存储元素，但提供了比序列索引更多的访问接口。

随机访问索引位于头文件<boost/multi_index/random_access_index.hpp>。

7.6.1 索引说明

随机访问索引的索引说明是random_access，它的类摘要如下：

```
template <typename TagList = tag<> >
struct random_access
{
  template<typename SuperMeta>
  struct index_class
  {
    typedef detail::random_access_index<...>  type;
  };
};
```

random_access与sequenced非常相像，同样不使用键提取器，只有一个标签模板参数。

7.6.2 类摘要

随机访问索引使用的类是detail::random_access_index，它是sequenced_index的超集，拥有sequenced_index的全部能力，类摘要如下：

```
class random_access_index
{
  public:
  typedef some_define value_type;
  typedef some_define iterator;
  typedef iterator const_iterator;
  ... //其他类型定义
```

```
    //赋值操作
    random_access_index& operator=(const random_access_index& x);
    void assign(InputIterator first,InputIterator last);
    void assign(size_type n,const value_type& value);

    //迭代器操作
    iterator begin();
    iterator end();
    iterator iterator_to(const value_type& x);

    //元素访问
    const_reference operator[](size_type n);          //随机访问接口
    const_reference at(size_type n)const;             //随机访问接口
    const_reference front()const;
    const_reference back()const;

    ...//其他同 sequenced_index 操作

    ...//各种比较操作符定义
};
```

random_access_index比sequenced_index多出了两个可以随机访问任意位置元素的operator[]和at()，其他的接口与sequenced_index相同，但操作的时间复杂度不同。

7.6.3 用法

random_access_index的用法几乎与sequenced_index一样，因为提供了随机访问的能力所以颇接近std::vector。不过它本质上还是链式容器，不像std::vector那样提供连续的元素存储，还能够使用push_front()在前段添加元素，看做是附加了部分std::vector功能的std::list比较合适。

示范随机访问索引用法的代码如下：

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/random_access_index.hpp>
using namespace boost::multi_index;                   //打开名字空间

int main()
{
    typedef multi_index_container<int,
        indexed_by<random_access<> >                  //随机访问索引，没有标签
```

```
    > mic_t;

using namespace boost::assign;
mic_t mic1 = (list_of(2),3,5,7,11);                    //初始化容器

assert(mic1[0] == 2);                                  //使用 operator[]
assert(mic1.at(2) == 5);使用 at()

mic1.erase(boost::next(mic1.begin(), 2));              //删除元素
assert(mic1[2] == 7);

mic_t mic2;
mic2.splice(mic2.end(), mic1);                         //使用链表的接合操作
assert(mic1.empty() && mic2.size() == 4);

push_front(mic1)(8), 10, 20, 16;                       //调用 push_front()

mic1.sort();                                           //内部的排序算法
mic2.sort();

mic1.merge(mic2);                                      //合并两个链表

for (mic_t::reverse_iterator iter = mic1.rbegin(); //逆序输出元素
    iter != mic1.rend(); ++iter)
{
    cout << *iter << ",";
}
}
```

代码的运行结果如下：

```
20,16,11,10,8,7,3,2
```

虽然random_access_index提供了随机访问元素的能力，但因为它的接口是常量性的，所以很多需要使用迭代器赋值操作的标准算法（如排序算法、替换算法）都无法使用：

```
std::sort(mic1.begin(), mic1.end());                   //编译错误
std::random_shuffle(mic1.begin(), mic1.end());         //编译错误
std::replace(mic1.begin(), mic1.end(), 2, 222);        //编译错误
```

7.7　有序索引

有序索引基于键提取器对元素进行排序，使用红黑树结构提供类似std::set和

std::multiset的有序集合访问接口。

有序索引位于头文件<boost/multi_index/ordered_index.hpp>。

7.7.1 索引说明

有序索引的索引说明包括ordered_unique和ordered_non_unique，前者不允许重复键（单键）而后者允许重复键（多键），两者的声明和接口均相同，故下面仅以ordered_unique为例。

ordered_unique的类摘要如下：

```
template<typename Arg1,typename Arg2=mpl::na,typename Arg3=mpl::na>
struct ordered_unique
{
  template<typename SuperMeta>
  struct index_class
  {
    typedef detail::ordered_index<...> type;
  };
};
```

ordered_unique有以下三个模板参数，但因为它使用了模板元编程技术，所以最少只要提供一个模板参数就可以工作：

- 第一个参数可以是标签或者键提取器，必须提供；
- 如果第一个参数是标签，那么第二个参数必须是键提取器；
- 最后一个参数是比较谓词对象，缺省是std::less<typename KeyFromValue::result_type>，即对键提取器的小于比较。

7.7.2 类摘要

有序索引使用的类是detail::ordered_index，接口类似std::set和std::multiset，类摘要如下：

```
template<typename KeyFromValue,typename Compare,...>
class ordered_index
{
public:
```

```
typedef some_define key_type;
typedef some_define value_type;
typedef some_define iterator;
typedef iterator    const_iterator;
... //其他类型定义

//赋值操作
ordered_index& operator=(const ordered_index& x);

//迭代器操作
iterator begin();
iterator end();
iterator iterator_to(const value_type& x);

//元素访问
std::pair<iterator,bool> insert(const value_type& x);
std::pair<iterator,bool> insert(iterator position,const value_type& x);
iterator erase(iterator position);

//有序相关操作
iterator find(const CompatibleKey& x)const;
iterator find(
  const CompatibleKey& x,const CompatibleCompare& comp)const;

size_type count(const CompatibleKey& x)const;
size_type count(const CompatibleKey& x,const CompatibleCompare& comp)const;

iterator lower_bound(const CompatibleKey& x)const;
iterator lower_bound(
  const CompatibleKey& x,const CompatibleCompare& comp)const;

iterator upper_bound(const CompatibleKey& x)const;
iterator upper_bound(
  const CompatibleKey& x,const CompatibleCompare& comp)const;

std::pair<iterator,iterator> equal_range(
  const CompatibleKey& x)const;
std::pair<iterator,iterator> equal_range(
  const CompatibleKey& x,const CompatibleCompare& comp)const;
```

```
  std::pair<iterator,iterator> range(
    LowerBounder lower,UpperBounder upper)const;

  ...//各种比较操作符定义
};
```

ordered_index的接口与std::set和std::multiset基本相同，但find()、count()、lower_bound()等涉及键比较的函数都有两种重载形式，其原理与侵入式容器的使用键操作值（参见 6.5.6 小节）方式类似，可以定义一个比较谓词函数对象CompatibleCompare使用兼容键比较，避免了构造大对象的成本。

ordered_index另一个特殊函数是range()，它使用两个函数对象简化了取上下界的操作（lower_bound和upper_bound）。

7.7.3 基本用法

有序索引的基本用法很简单，与std::set和std::multiset没有太多区别，当然还要注意不能使用迭代器来修改元素，因为多索引容器的元素是不可变的。

下面的代码示范了单键有序索引的简单用法：

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/ordered_index.hpp>   //有序索引
#include <boost/multi_index/key_extractors.hpp>
using namespace boost::multi_index;

int main()
{
  using namespace boost::assign;

  typedef multi_index_container<int,            //元素类型为 int
      indexed_by<                               //索引说明列表
          ordered_unique<identity<int>>         //有序单键索引，使用 identity
      >
  > mic_t1;

  mic_t1 mic1;
  insert(mic1)(2), 3, 5, 7, 11;                 //插入不允许重复的元素

  assert(mic1.size() == 5);
```

```
    assert(!mic1.insert(3).second);                //重复元素无法插入
    assert(mic1.find(7) != mic1.end());
}
```

接下来的代码示范了多键有序索引的用法，读者需注意键提取器的使用：

```
int main()
{
    using namespace boost::assign;

    typedef multi_index_container<string,       //元素类型为 string
        indexed_by<                              //索引说明列表
            ordered_non_unique<                  //有序多键索引，使用字符串长度作为键
                BOOST_MULTI_INDEX_CONST_MEM_FUN(string, size_t, size)>
        >
    > mic_t2;

    mic_t2 mic2;
    insert(mic2)("111")("22")("333")("4444");  //注意，有重复元素
    assert(mic2.count(3) == 2);                 //两个重复元素

    //使用 equal_range()输出重复的元素
    BOOST_FOREACH(const string& str, mic2.equal_range(3))
    {   cout << str << ",";}
}
```

这里的有序索引定义略微复杂一些。虽然元素类型是字符串string，但我们并没有使用字符串本身作为键（identity<string>），而是使用字符串长度作为键，键提取器是const_mem_fun，调用了string的const成员函数size()。

因为键的特殊性，虽然我们向容器中插入了四个字面值不同的字符串，但实际上"111"和"333"这两个元素是重复的，因为它们的长度都是 3，所以调用count()和equal_range()能够看到两个重复的元素。

7.7.4　高级用法

本小节使用person类（在 7.2 小节有声明）作为容器的元素，讨论有序索引的两种高级用法：兼容键比较和获得范围区间。

兼容键比较

所谓兼容键（compatible key），是指不同于索引说明中键本身的一个类型，但它的比较效果是与键相同的。例如，对于person类来说，identity<person>定义键类型为

person，但它的比较操作符operator<使用的是类型为int的m_id成员变量，所以int就是它的兼容键。很显然，构造一个int类型的成本要比构造一个person类型的成本低很多，效率上自然会有很大提升。

为了使用兼容键比较函数，我们需要为person类自定义一个与int类型比较的谓词，比较关系应与容器的比较谓词一致（在这里是小于关系）：

```
struct compare_by_id                //比较 person 对象和 id
{
  typedef bool result_type;
  bool operator()(const person& p, int id) const
  {   return p.m_id < id;}
  bool operator()(int id, const person& p) const
  {   return id < p.m_id ;}
};
```

这样我们就可以仅使用兼容键类型而不必使用整个元素（键类型）来执行比较操作了，提高了效率，示范代码如下：

```
int main()
{
  using namespace boost::assign;

  typedef multi_index_container<person,
      indexed_by<
          ordered_unique<identity<person>>        //有序单键索引
      >
  > mic_t;

  mic_t mic;
  insert(mic)
      (person(2, "agent", "smith"))               //插入四个元素
      (person(20, "lee", "someone"))
      (person(1, "anderson", "neo"))
      (person(10, "lee", "bruce"));

  //构造一个大对象执行比较操作，成本很高
  assert(mic.count(person(1, "abc","xby")) == 1);

  //使用自定义的兼容键比较谓词
  assert(mic.count(1, compare_by_id()) == 1);
  assert(mic.find(10, compare_by_id()) != mic.end());
```

```
}
```

获取范围区间

成员函数range()是一个模板函数，它的参数（LowerBounder和UpperBounder）是两个以键为参数的谓词函数或函数对象，用于确定区间的下界和上界，相当于a < x && x < b，比直接使用lower_bound()和upper_bound()要方便直观的多。

例如，我们为person的id定义一个 2 <= p.m_id < 20 的左闭右开区间，使用lower_bound()和upper_bound()的用法如下：

```
...//多索引容器的声明和数据插入同前
//获得 id>=2 的下界
mic_t::iterator l = mic.lower_bound(2 , compare_by_id());
//获得 id>=20 的下界，即<20 的上界
mic_t::iterator u = mic.lower_bound(20, compare_by_id());
//foreach 循环，使用 make_pair 构造一个迭代器区间输出元素
BOOST_FOREACH(const person& p,
    std::make_pair(l, u))
{
    cout << p.m_id << ":"               //输出两个元素
        << nameof(p) << endl;           //2:agent smith 和 10:lee bruce
}
```

这段代码中的两个lower_bound()很令人迷惑，即使是有经验的STL程序员也很难把握区间的端点条件，很容易出错。

使用有序索引的range()函数就简单多了，所需的上下界函数对象可简单清晰地实现如下：

```
struct lower_bounder                    //下界函数对象，定义 p >= 2
{
  typedef bool result_type;
  bool operator()(const person& p)
  {  return p.m_id >= 2;}
};
struct upper_bounder                    //上界函数对象，定义 p< 20
{
  typedef bool result_type;
  bool operator()(const person& p)
  {  return p.m_id < 20;}
```

```
};
```

然后调用range()函数就可以轻易且精确地获得这个区间：

```
BOOST_FOREACH(const person& p,
        mic.range(lower_bounder(), upper_bounder()))
{...}                               //循环体省略
```

如果我们经常执行获取区间的操作，那么编写大量类似的上下界谓词会很烦琐，这时我们可以使用boost.bind（可参考推荐书目[1]）来组合已有的标准函数对象。例如，lower_bounder和upper_bounder等价的bind表达式是：

```
BOOST_FOREACH(const person& p,
        mic.range(
        bind(std::greater_equal<int>(),      //绑定标准大于等于函数对象
            bind(&person::m_id, _1), 2),     //再绑定取成员变量
        bind(std::less<int>(),               //绑定标准小于函数对象
            bind(&person::m_id, _1), 20)     //再绑定取成员变量
        ))
{...    }                                    //循环体省略
```

boost.lambda库可以就地生成匿名函数对象，它要比bind表达式略微简化一些（需要包含头文件<boost/lambda/lambda.hpp>）。

```
BOOST_FOREACH(const person& p,          \
    mic.range(                          \
    boost::lambda::_1 >= person(2, "no", "no"),\
    boost::lambda::_1 <  person(20, "no", "no")\
    ))
{...    }                                    //循环体省略
```

为了支持无限制上界和无限制下界，有序索引还定义了一个特别的谓词函数unbounded，它可以用在区间的任意一个端点，表示该端点无限制，例如：

```
mic.range(lower_bounder(), unbounded)        //p.m_id >= 2
mic.range(unbounded, upper_bounder(),)       //p.m_id <  20
mic.range(unbounded, unbounded)              //所有元素
```

7.8 散列索引

散列索引基于键提取器对（元素的）键进行散列，提供类似boost::unordered_set

和boost::unordered_multiset的无序集合接口。

散列索引位于头文件<boost/multi_index/hashed_index.hpp>。

7.8.1 索引说明

散列索引的索引说明包括hashed_unique和hashed_non_unique，前者不允许重复键而后者允许重复键，两者的声明和接口均相同，故下面仅以hashed_unique为例。

hashed_unique的类摘要如下：

```
template<typename Arg1,typename Arg2,typename Arg3,typename Arg4>
struct hashed_unique
{
  template<typename SuperMeta>
  struct index_class
  {
    typedef detail::hashed_index<...> type;
  };
};
```

hashed_unique有以下四个模板参数，使用了与ordered_unique同样的模板元编程技术，最少只要提供一个模板参数就可以工作：

- 第一个参数可以是标签或者键提取器，必须提供；
- 如果第一个参数是标签，那么第二个参数必须是键提取器；
- 键提取器后的参数是散列函数对象，缺省是boost::hash<typename KeyFromValue::result_type>；
- 最后一个参数是相等比较谓词对象，缺省是std::equal_to<typename KeyFromValue::result_type>。

7.8.2 类摘要

散列索引使用的类是hashed_index，接口类似boost::unordered_set和boost::unordered_multiset，类摘要如下：

```
template<typename KeyFromValue,typename Hash,typename Pred,...>
class hashed_index
{
```

```
public:
  typedef some_define key_type;
  typedef some_define value_type;
  typedef some_define iterator;
  typedef iterator const_iterator;
  ... //其他类型定义

  //赋值操作
  hashed_index& operator=(const hashed_index& x);

  //迭代器操作
  iterator begin();
  iterator end();
  iterator iterator_to(const value_type& x);

  //元素访问
  iterator find(const CompatibleKey& x)const;
  iterator find(
    const CompatibleKey& x,
    const CompatibleHash& hash,const CompatiblePred& eq)const;

  size_type count(const CompatibleKey& x)const;
  size_type count(
    const CompatibleKey& x,
    const CompatibleHash& hash,const CompatiblePred& eq)const;

  std::pair<iterator,iterator> equal_range(const CompatibleKey& x)const;
  std::pair<iterator,iterator> equal_range(
    const CompatibleKey& x,    const CompatibleHash& hash,const CompatiblePred&
eq)const;

  ... //各种比较操作符定义
};
```

hashed_index与boost::unordered_set的接口类似，因为是无序的，所以不提供成员函数lower_bound()、upper_bound()和range()。此外hashed_index还有一些散列容器相关的特殊成员函数，如桶数量、负载因子等，本章从略。

7.8.3 用法

散列索引用起来就像是boost::unordered_set和boost::unordered_multiset，它们都遵循了std::tr1 标准草案。示范散列索引用法的代码如下：

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/key_extractors.hpp>
using namespace boost::multi_index;

int main()
{
  using namespace boost::assign;

  typedef multi_index_container<person,
      indexed_by<
          hashed_unique<mi::identity<person>>        //散列单键索引
          >
      > mic_t;

  mic_t mic;
  insert(mic)
      (person(2, "agent", "smith"))                  //插入元素
      (person(1, "anderson", "neo"))

  assert(mic.size() == 4);
  assert(mic.find(person(1, "anderson", "neo"))!= mic.end());
};
```

散列索引同样可以使用兼容键来查找元素，但它需要使用两个函数对象用于散列和相等比较，比有序索引要多做一些工作。

由于person类散列使用了m_fname和m_lname，相等比较使用了m_id，涉及的因素较多，所以我们使用一个boost::tuple来定义兼容键：

```
//用 tuple 组合 3 个类型定义兼容键
typedef boost::tuple<int, string, string> hash_key_t;

//定义散列函数对象，算法与 person 的一致
struct hash_func
{
  typedef size_t result_type;
  size_t operator()(const hash_key_t& k) const       //必须是 const
  {
      size_t seed = 2011;
```

```
        hash_combine(seed, k.get<1>());
        hash_combine(seed, k.get<2>());
        return seed;
    }
};

//定义相等比较函数对象
struct equal_func
{
    typedef bool result_type;
    bool operator()(const hash_key_t& k, const person& p) const
    {   return k.get<0>() == p.m_id;}
    bool operator()(const person& p, const hash_key_t& k) const
    {   return k.get<0>() == p.m_id;}
};
```

这样我们就可以使用兼容键在散列索引中查找元素了：

```
assert(mic.count(make_tuple(1, "anderson", "neo"),
        hash_func(), equal_func()) == 1);
assert(mic.find(make_tuple(10, "lee", "bruce"),
        hash_func(), equal_func())!= mic.end());
```

7.9 修改元素

multi_index库中所有索引的迭代器接口都是常量性的，不允许用户直接修改，这是因为一个索引的元素变动操作可能会导致其他索引不一致，破坏整个多索引容器的结构。

但并不是说多索引容器里的元素就是不可修改的，multi_index为此使用了另外的机制：所有的索引都提供两个专门用于修改元素的成员函数：replace()和modify()，有序索引和散列索引还有一个特别的修改键的成员函数modify_key()，使用这些操作可以保持多索引容器的状态不被破坏。

7.9.1 替换元素

成员函数replace()可以替换一个有效迭代器位置上元素的值，其他索引保持同步更新。multi_index库为replace()操作提供了强异常安全保证，即使发生异常多索引容器也会保持不变，所有索引及相关的迭代器和引用也保持不变。

replace()的声明如下：

```
bool replace(iterator position,const value_type& x);
```

迭代器通常可以使用find()算法或者find()成员函数获得，如果使用iterator_to()则需要小心，使用元素的等价拷贝获得的是一个无效迭代器，用于replace()会发生运行时异常。

示范replace()用法的代码如下：

```
typedef multi_index_container<person,              //定义一个三索引的容器
    indexed_by<
        ordered_unique<mi::identity<person>>               //单键有序索引
        ordered_non_unique<                                //有序多键索引
            member<person, string, &person::m_fname>>,     //使用成员变量
        hashed_unique<mi::identity<person>>                //散列单键索引
    >
> mic_t;

mic_t mic;
insert(mic)
    (person(2, "agent", "smith"))                      //插入 4 个元素
    (person(20, "lee", "someone"))
    (person(1, "anderson", "neo"))
    (person(10, "lee", "bruce"));

BOOST_AUTO(pos, mic.find(20, compare_by_id()));        //查找 id 为 20 的元素
assert(pos != mic.end());

mic.replace(pos, person(20, "lee", "long"));           //替换这个元素
assert(pos->m_lname == "long");
```

执行替换操作时可以任意修改元素的值，元素的键也可以被修改，例如：

```
mic.replace(pos, person(15, "lee", "long"));           //修改元素的键
```

但如果修改后的元素与索引约束发生冲突则会替换失败，函数返回false：

```
assert(!mic.replace(pos, person(2, "lee", "someone")));  //索引 0 冲突
assert(!mic.replace(pos, person(10, "lee", "bruce")));   //索引 2 冲突
```

7.9.2 修改元素

replace()成员函数提供了强异常安全保证，但操作的代价较高，因为在替换时我们必须构造一个完整的临时元素对象，很多时候是不必要的。

modify()是另外一种修改元素的方法，它使用一个被称为修改器的函数或函数对象，接受一个元素的引用作为参数，可以只变动元素的某个成分，是轻量级的修改元素的方法。

修改

modify()的声明如下：

```
template<typename Modifier>
bool modify(iterator position,Modifier mod);
```

modify()有两个参数，第一个是要修改的迭代器位置，含义与replace()相同，第二个是修改器对象，它执行元素的修改操作。

例如，我们想要修改person的各个成员变量，可以编写这样的修改器函数和函数对象：

```
void modify_id(person& p, int id)                    //修改m_id
{
  p.m_id = id;
}
void modify_fname(person& p, const string& fname)    //修改m_fname
{
  p.m_fname = fname;
}
struct modify_lname                                  //修改m_lname
{
  string m_lname;
  modify_lname(const string& lname):                 //构造函数传入要变更的新值
      m_lname(lname){}
  void operator()(person& p)
  {
      p.m_lname = m_lname;
  }
};
```

modify_id()和modify_fname()有两个参数，所以在传递给modify()函数时需要使用bind绑定修改值转换为单参函数对象，而modify_lname因为是函数对象，所以可以使用构造函数传入修改值：

```
  mic.modify(pos, bind(&modify_id, _1, 15));         //修改m_id
  assert(pos->m_id == 15);

  mic.modify(pos, bind(&modify_fname, _1, "mike")); //修改m_fname
  assert(pos->m_fname == "mike");

  mic.modify(pos, modify_lname("david"));           //修改m_lname
```

```
assert(pos->m_lname == "david");
```

回滚

modify()避免了构造临时对象的成本，执行效率高，但也有不如replace()的地方：它不能保证操作的安全性，如果元素修改后与索引的约束发生冲突修改将失败，而元素会被删除！请看下面的代码：

```
//找到 id 为 20 的元素
BOOST_AUTO(pos, mic.find(20, compare_by_id()));

//修改 id 为 1，与索引 0 的约束(ordered_unique)发生冲突，修改失败，元素被删除
assert(!mic.modify(pos, bind(&modify_id, _1, 1)));

//此时元素已被删除，无法找到
assert(mic.size() == 3);
assert(mic.find(20, compare_by_id()) == mic.end());
```

为了避免这样的“灾难”发生，modify()提供了一个类似数据库“回滚”机制(rollback)的重载形式，允许用户使用一个“回滚”函数或函数对象在修改失败时恢复原有的值，这种形式的modify()函数原型如下：

```
template<typename Modifier,typename Rollback>
bool modify(iterator position,Modifier mod,Rollback back);
```

回滚机制的modify()用法如下：

```
assert(!mic.modify(pos,
    bind(&modify_id, _1, 1),              //修改 id 为 1，发生冲突
    bind(&modify_id, _1, 9999)));         //回滚操作，把 id 改为 9999，不是很好
assert(mic.size() == 4);
assert(mic.find(100, compare_by_id()) != mic.end());
```

上面的代码不是解决问题的最佳答案，因为固定的id还有可能造成冲突，但这时索引却一无所知，可能会导致索引混乱（读者可以试着把 9999 改为已有的id试验一下）。正确的做法是先使用迭代器获得要修改的原值，然后把原值作为回滚的参数：

```
int tmp = pos->m_id;                      //修改前先保存原值
assert(!mic.modify(pos,
    bind(&modify_id, _1, 1),              //修改器修改
    bind(&modify_id, _1, tmp)));          //如果修改失败那么回滚恢复原值
```

7.9.3 修改键

有序索引和散列索引拥有一个特别的修改函数modify_key()，它可以直接修改索引使用的键而不是元素本身，是modify()的特化版本。使用modify_key()要求索引的键提取器必须是可写的。

modify_key()的声明如下：

```
template<typename Modifier>
bool modify_key(iterator position,Modifier mod);
template<typename Modifier,typename Rollback>
bool modify_key(iterator position,Modifier mod,Rollback back);
```

modify_key()的修改器操作的不是元素本身的类型，而是键类型，所以为了修改使用字符串作为键的索引我们要重新定义一个函数：

```
void modify_str(string& str, const string&new_str)      //修改字符串
{
  str = new_str;
}
```

然后我们就可以获取索引使用modify_key()来修改键：

```
BOOST_AUTO(&index, mic.get<1>());                        //获得索引
BOOST_AUTO(pos, index.find("agent"));                    //查找元素

index.modify_key(pos, bind(&modify_str, _1, "virus")); //修改键
assert(pos->m_fname == "virus");
```

因为键通常都是简单类型，所以使用lambda可以进一步简化修改的代码：

```
index.modify_key(pos, boost::lambda::_1 = "virus");
```

同样的，如果modify_key()修改键后导致索引冲突元素也会被立即删除，为了安全起见应该使用回滚操作：

```
string tmp = pos->m_fname;
index.modify_key(pos,
    boost::lambda::_1 = "virus",                         //修改键
    boost::lambda::_1 = tmp);                            //失败则恢复原值
```

7.10 多索引容器

经过了前面数节对键提取器和索引的研究，我们终于来到了真正的多索引容器multi_index_container面前，下面就来看看multi_index_container的真面目。

7.10.1 类摘要

在7.3.6小节已经列出了multi_index_container的前置声明，下面是它的类摘要：

```
template<
  typename Value,                                //元素类型
  typename IndexSpecifierList=indexed_by<...>,   //索引说明列表
  typename Allocator=std::allocator<Value> >     //内存分配器
class multi_index_container:
     public detail::multi_index_base_type<...>::type
{
public:

  //构造函数、赋值操作
  multi_index_container(InputIterator first,InputIterator last);
  multi_index_container(const multi_index_container& x);
  multi_index_container& operator=(const multi_index_container& x);

  //索引类型定义
  template<int N> struct nth_index;
  template<typename Tag>  struct index;

  //获取索引操作
  template<int N>
  typename nth_index<N>::type& get();
    template<typename Tag>
  typename index<Tag>::type& get()

  //投射操作
  template<int N,typename IteratorType>
    typename nth_index<N>::type::iterator project(IteratorType it);
  template<typename Tag,typename IteratorType>
    typename index<Tag>::type::iterator project(IteratorType it);
};
```

multi_index_container本质上是一个索引的容器，它本身只负责管理索引，各个索引负责元素的管理。

为了方便使用容器，multi_index_container使用模板元编程技术实现了从第一个索引继承（public detail::multi_index_base_type<...>::type），获得了它的所有接口和能力，因此我们可以直接以容器的方式使用第一个索引。

get<>()是multi_index_container最重要的成员函数，它有两种重载形式，分别可以使用整数序号或者类型标签来获取索引，因此索引也有两个类型，分别是nth_index<N>和index<tag>。这两个类型实际上是元函数，需要使用::type的形式才能获得真正的索引类型，不过通常我们可以使用BOOST_AUTO来简单地避免这个类型声明的问题。

project<>()用来在多个不同的索引之间转换迭代器，可以把一个索引的迭代器投射到另一个索引的迭代器，而这两个迭代器指向的是同一个元素，这可以方便我们以一个索引查找元素再改用另一个索引操作元素。

multi_index_container是可序列化的，详细信息可参见第 9.5.6 小节。如果不需要使用序列化能力，可以定义宏BOOST_MULTI_INDEX_DISABLE_SERIALIZATION，这样将禁用序列化代码，可加快编译速度。

7.10.2 用法

multi_index_container真正属于自己的操作不是很多，因为它主要负责管理索引，使用get<>()获得索引后由索引来操作元素。

作为示范，这里我们综合使用之前介绍的所有键提取器和索引，定义一个持有八个索引的容器。这样的一个复杂多索引容器，如果直接写出索引说明列表是非常庞大的，所以我们必须使用typedef进行简化。

首先是序列索引和随机访问索引的定义，使用了索引标签：

```
typedef sequenced<tag<int, struct seq_idx> >            idx_sf0;
typedef random_access<tag<struct rnd_idx, string> >     idx_sf1;
```

然后是三个有序索引，分别使用了identity、member和const_mem_fun键提取器，最后一个ordered_non_unique索引还使用std::greater<>谓词改变了排序准则：

```
typedef ordered_unique<mi::identity<person>> idx_sf2;   //有序单键索引
typedef ordered_non_unique<                             //有序多键索引
```

```
        BOOST_MULTI_INDEX_MEMBER(person, string, m_fname)
                                > idx_sf3;
    typedef ordered_ unique<                            //有序单键索引
        BOOST_MULTI_INDEX_CONST_MEM_FUN(
            person, const string&, first_name),
        std::greater<const string>                      //使用大于比较排序
                              > idx_sf4;
```

最后是三个散列索引，使用了member、global_fun和自定义键提取器：

```
    typedef hashed_unique<                //散列单键索引
        BOOST_MULTI_INDEX_MEMBER(person, string, m_lname)
                                > idx_sf5;
    typedef hashed_non_unique<            //散列多键索引
        global_fun<const person&, string, &nameof>
        > idx_sf6;
    typedef hashed_unique<                //散列多键索引
        person_name>           idx_sf7;   //使用自定义键提取器
```

读者需注意：以上八个索引中我们没有使用mem_fun键提取器，这是因为容器存储的是元素本身而不是指针，如果使用mem_fun会因为从常量中无法提取键而导致编译失败（参见 7.4.5 小节）。

有了这些索引的类型定义，多索引容器的定义就简单多了：

```
    typedef multi_index_container<person,      //定义多索引的容器
        indexed_by<
            idx_sf0, idx_sf1,                  //序列索引和随机访问索引
            idx_sf2, idx_sf3, idx_sf4,         //有序索引
            idx_sf5, idx_sf6, idx_sf7          //散列索引
        >
    > mic_t;
```

对于这个多索引容器，我们可以使用以下八个完全不同的接口访问它：

- 像 std::list 一样的双向链表；
- 像 std::vector 一样的随机访问序列；
- 基于 m_id 从小到大排序的不允许重复的有序集合；
- 基于 m_fname 从小到大排序的允许重复的有序集合；

- 基于 m_fname 从大到小排序的不允许重复的有序集合；
- 基于 m_lname 的不允许重复的无序集合；
- 基于 m_fname+m_lname 的允许重复的无序集合；
- 基于 m_fname+m_lname 的不允许重复的无序集合。

由于这个容器使用的索引比较多，所以相互之间的制约也就错综复杂，单纯的读操作不会有什么影响，如果执行插入、修改等涉及元素变动的操作就必须小心，不能违反任何一个索引的约束。例如，idx_sf3 允许m_fname重复，但idx_sf4 却不允许m_fname重复，这样实际上是把idx_sf3 的non_unique效果“抵消”了。代码如下所示：

```
mic_t mic ;

using namespace boost::assign;
push_back(mic)
    (person(2, "agent", "smith"))       //插入四个元素
    (person(20, "lee", "someone"))      //插入成功
    (person(1, "anderson", "neo"))      //插入成功
    (person(10, "lee", "bruce"));       //因为 m_fname 重复所以插入失败

assert(mic.size() == 3);                //最终只插入了 3 个元素
```

示范成员函数project()用法的代码如下：

```
BOOST_AUTO(&idx1, mic.get<rnd_idx>());              //获得随机访问索引
BOOST_AUTO(&idx2, mic.get<2>());                    //获得有序索引

BOOST_AUTO(pos, idx2.find(1, compare_by_id()));     //在有序索引中查找元素
BOOST_AUTO(pos2, mic.project<string>(pos) );        //投射到随机访问索引

assert(pos2 == idx1.iterator_to(idx1[2]));          //使用 iterator_to()
```

这段代码中我们使用有序索引查找m_id为 1 的元素，然后把迭代器投射到随机访问索引上，因为我们已经知道这个元素的位置，所以直接使用operator[]获得元素的引用，再用iterator_to()转换得到迭代器，验证了投射的正确性。

mem_fun键提取器可以用在容纳指针或智能指针的多索引容器里，例如：

```
typedef shared_ptr<person> person_ptr;          //存储智能指针

typedef hashed_unique<                          //定义一个新的散列单键索引
```

```
        BOOST_MULTI_INDEX_MEM_FUN(                  //使用 mem_fun
        person, string&, last_name)
    > idx_sf8;
    typedef multi_index_container<person_ptr,      //定义多索引容器
        indexed_by<idx_sf8> > mic_t;

    mic_t mic ;
    mic.insert(make_shared<person>(2, "agent", "smith"));
```

7.11 组合索引键

类似数据库里的联合主键概念，有的时候我们仅使用一个键对元素排序可能还不够，需要同时基于多个键来查找元素，这时我们就要用到multi_index库提供的组合索引键composite_key的功能。

composite_key可以把多个键提取器组合为一个新的键供索引使用，能够提供更多的灵活性，它位于头文件<boost/multi_index/composite_key.hpp>。

7.11.1 类摘要

composite_key是一个只读键提取器，类摘要如下：

```
template<typename Value,typename KeyFromValue0,...>
struct composite_key : private tuple<...>
{
  typedef Value                                    value_type;
  typedef composite_key_result<composite_key> result_type;

  result_type operator()(const value_type& x)const;
  ...//其他 operator()定义
};
```

composite_key基于boost.tuple实现对多个键提取器的组合，它的第一个模板参数是元素类型Value，可以跟着一个或多个组合的键提取器。这些键提取器可以是identity、member、const_mem_fun等任意的键提取器，但必须也以Value作为元素类型。当前multi_index库支持组合最多 10 个键提取器（低版本的VC只支持 5 个）。

composite_key的operator()返回类型是composite_key_result，它是一个非常简单的struct，重载了标准比较谓词equal_to、less、greater和boost的hash操作，

并且支持与tuple的比较操作，可以像基本类型一样使用。

composite_key_result的比较规则与tuple相同，依据字典序，即依据键的顺序逐个比较。

7.11.2 用法

composite_key可以用于有序索引和散列索引，由于它自身定义已经很复杂，如果直接在multi_index_container中定义会进一步增加多索引容器的复杂性，所以最好使用typedef，同时注意缩进格式。

我们先使用composite_key定义一个基于m_id和m_fname的组合索引键，注意元素类型使用了指针，意味着我们要在容器中存储指针类型：

```
typedef composite_key<person*,          //元素类型为指针
    BOOST_MULTI_INDEX_MEMBER(person, int, m_id),
    BOOST_MULTI_INDEX_MEMBER(person, string, m_fname)
> comp_key;
```

使用组合键的多索引容器定义如下，元素类型与组合键相同，也是指针：

```
typedef multi_index_container<
    person*,                            //元素类型为指针
    indexed_by<
        ordered_unique<comp_key>        //使用组合键
    >
> mic_t;
```

为了避免手工删除指针的麻烦，我们可以使用指针容器来存储元素，把多索引容器当做指针容器的一个视图来使用：

```
ptr_vector<person> vec;
using namespace boost::assign;
ptr_push_back(vec)                      //使用 assign 库插入动态创建的元素
    (2, "agent", "smith")
    (1, "anderson", "neo")
    (1, "the one", "neo");              //注意，id 有重复

mic_t mic;
BOOST_FOREACH(person& p, vec)           //插入指针元素到多索引容器
{
    mic.insert(&p);
}
```

```
BOOST_FOREACH(person * const  p, mic)   //顺序输出多索引容器内的元素
{
    cout << p->m_id << ":"
        << p->m_fname << ",";
}
```

请读者注意组合键的排序准则，这里是基于m_id和m_fname，与单纯的identity <person>等不同，只有m_id和m_fname都相同时才能判断为重复。

使用find()、count()等涉及组合键的查找操作时我们需要使用tuple来构造查找值，因为composite_key_result不能够直接创建：

```
assert(mic.count(make_tuple(1, "anderson")) == 1);
assert(mic.find(make_tuple(2, "agent")) != mic.end());
```

在有序索引中我们也可以仅顺序指定部分键值，这样索引将执行“模糊查找”，仅对指定的查找值执行比较：

```
assert(mic.count(make_tuple(1)) == 2); //仅指定一个键
```

如果仅使用第一个键，那么multi_index库允许我们不使用tuple，直接使用值：

```
assert(mic.count(1) == 2);              //仅指定一个键，不必用 tuple
```

对于散列索引我们不能指定部分键值，因为散列必须基于所有的键。

7.11.3 辅助工具

单单把键组合起来可能还不够，我们有时还想对组合键做更多的定制工作，比如对一个键执行升序排序而对另一个键执行降序排序，multi_index库为此提供了多个操作composite_key_result的比较谓词和散列函数对象。

三个基本的函数对象可以组合基于单个键的比较谓词或散列函数对象任意定制排序准则。

```
template<typename Pred0,...,typename Predn>
struct composite_key_equal_to;
template<typename Compare0,...,typename Comparen>
struct composite_key_compare;
template<typename Hash0,...,typename Hashn>
struct composite_key_hash;
```

例如，我们可以对person类的m_id升序而对m_fname降序，定制排序准则如下：

```
typedef composite_key_compare<
    std::less<int>,             //m_id 升序
    std::greater<string>        //m_fname 降序
> comp_key_compare;
```

多索引容器的定义需要增加组合比较谓词的定义：

```
typedef multi_index_container<
    person*,
    indexed_by<
        ordered_unique<
            comp_key,           //组合键
            comp_key_compare    //组合比较谓词
        >
    >
> mic_t;
```

为了方便组合比较谓词的使用，multi_index库又定义了四个简化的函数对象，使用标准库的equal_to、less、greater和boost的hash操作所有键：

```
template<typename CompositeKeyResult>
struct composite_key_result_equal_to;
template<typename CompositeKeyResult>
struct composite_key_result_less;
template<typename CompositeKeyResult>
struct composite_key_result_greater;
template<typename CompositeKeyResult>
struct composite_key_result_hash;
```

例如，对m_id和m_fname降序排列定义如下：

```
typedef multi_index_container<
    person*,
    indexed_by<
        ordered_unique<
            comp_key,
            composite_key_result_greater<comp_key::result_type>
        >
    >
> mic_t;
```

注意composite_key_result_greater的用法，它需要使用一个composite_key_result作为模板参数，我们必须使用comp_key::result_type来获得这个类型，它实际上是composite_key_result<comp_key>。

7.12　总结

本章我们讨论了Boost中的多索引容器库multi_index，它提供了一个可以同时持有多个访问接口的容器multi_index_container，很适合需要以多种检索方式操作大量数据的情形，可以显著地提高性能，通常要比单纯使用标准容器再搭配算法要好很多。

multi_index_container的用法比较复杂，它的核心是索引说明列表，可以用index_by结构组合六种索引说明和五种键提取器，从而定义任意的索引方式。虽然multi_index_container提供了极大的灵活性，但付出的成本是代码复杂难懂，对于不熟悉多索引容器的人来说维护较困难，使用它需要慎重考虑这些程序之外的成本。

因为多个索引之间的制约关系，保持各个索引的正确性是一项必要的工作。multi_index_container要求元素不能被随意修改，迭代器接口都是常量性的，这样特意的设计可以很大程度上保持多索引容器的稳定。另一方面，multi_index_container又提供了replace()、modify()和modify_key()三个成员函数，可以使用函数或函数对象安全地修改元素全部或部分的值。

multi_index_container可以很好地替代标准容器vector、list、set和multiset，但替代映射容器map和multimap则比较麻烦，必须手工定义一个pair类作为容器的元素。这时我们可以考虑使用boost.bimap，它是一个基于multi_index库实现的双向映射容器，具有类似map的接口，同时又有multi_index_container的一些特性（在推荐书目[1]中有介绍）。

本章并没有完全覆盖multi_index的所有内容，也没有对各种索引的时间复杂度进行分析，读者可以在今后的实际工作中深入研究multi_index的更多用法。

第 8 章

流处理

流处理库（iostream）是 C++“出生”时即搭配的“标准库”，历史悠久，用于为 C++提供基于流式的输入输出功能（并不限于控制台）。很多人对 C++流处理的认识通常仅限于 cin、cout，把它当做是 C 语言 scanf()、printf()的 C++等价物来看待，作用也通常是打印日志或调试用语句等“小儿科”应用。在图形界面和应用程序框架“泛滥成灾”的如今，iostream 更是几乎被打入了“冷宫”，鲜有程序员把关注的目光投在它身上。

boost.iostreams 库重新挖掘了这颗“沉睡的珠宝”，令流处理再度回到大众的视野。它基于标准库的 I/O 流框架提供了富有弹性且易于使用的流式处理机制，使 C++处理数据更加简单、方便和高效。

通过本章的讨论，希望读者能够重新审视 iostream，给予它应有的位置，而不是仅仅把它当做是控制台的输入输出。

8.1　概述

本节将简要介绍标准库的流处理和 boost.iostreams 库的基本情况，不以精确描述或评判为目的，仅使读者对流处理有个大致的了解。

8.1.1　标准库的流处理

C++中将输入输出视为“流”（stream），即数据在其中“流动”的序列，数据处理即在流动中完成操作。流处理的数据通常是字符类型（char 或 wchar_t），但也可使用模板参数指定处理任意的类型，因此，流是一套完整的数据处理框架。

根据处理数据的方向，流可分为输入流和输出流两大类。输入是指从流中输入，而不是向流输入，类似于可读迭代器的概念；输出是指向流输出，而不是从流输出，类似于可写迭代器的概念。

C++中 stream 的核心类体系结构图如图 8-1 所示：

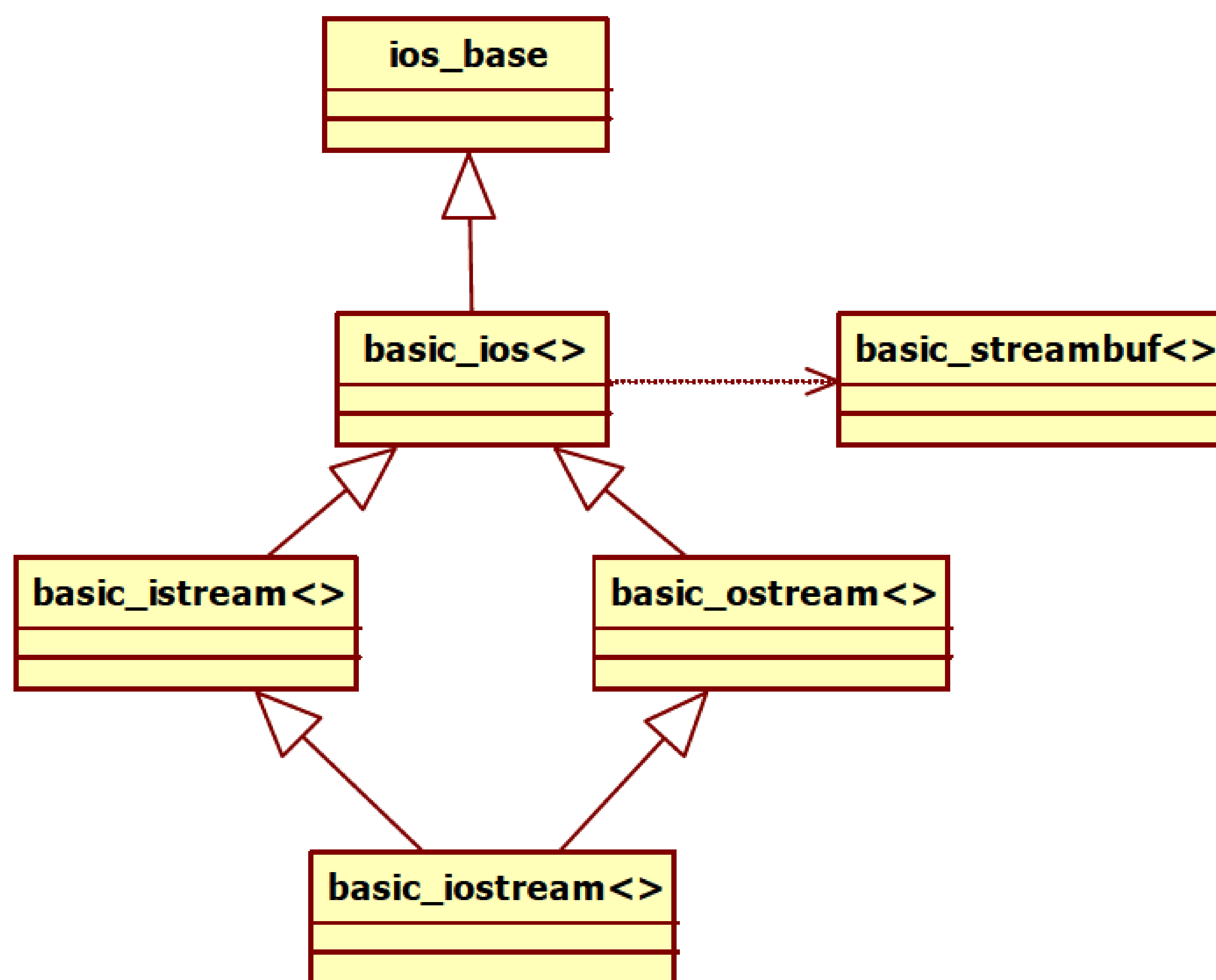


图 8-1 stream 核心类体系结构图

其中，ios_base 是流的基类，它的子类 basic_ios<>使用模板参数规定了流处理的数据类型和特性（traits），并依赖 basic_streambuf<>完成实际的读写操作。basic_istream<>和 basic_ostream<>分别从 basic_ios<>虚继承，定义了输入输出流（读写流），最后的 basic_iostream<>实现了既可读又可写的双向流。

我们常用的 cin、cout 的类型分别是 istream 和 ostream，而这两个类型则分别是 basic_istream<>和 basic_ostream<>的模板特化形式，即：

```
typedef basic_istream<char, char_traits<char> > istream;
typedef basic_ostream<char, char_traits<char> > ostream;
```

stream 重载了右移操作符（>>）和左移操作符（<<），重新定义为输入输出操作符（或称读取/写入、提取/插入操作符），可以用很简单的方式从流中读取数据或者向流中写入数据。此外，stream 还提供了大量的成员函数和操控器能够更细致地操作数据，例如 get()、put()、getline()、read()、write()、endl()等。

后续的讨论中我们将把精力集中于流的数据处理部分，而忽略数据的格式表述功能。流

的数据类型使用最常见的 char，但个别例子会使用特殊的数据类型。

8.1.2 Boost 的流处理

boost.iostreams 库建立在标准库的 IO 流框架基础之上，它定义了 device、source、sink、filter 等新的流处理概念，构造了一套全新的、更易于使用和定制的流处理框架，几乎可以用流来处理任何数据，例如字符串处理、base64 编解码、zip 压缩解压缩、数据的加密解密等。

总的来说，iostreams 库包含如下的组成部分：

- 设备（device）概念 ：包括流的起点 source（源设备）和流的终点 sink（接收设备），两者统称为设备；
- 过滤器（filter）概念：包括读取时处理数据的输入过滤器和写入时处理数据的输出过滤器；
- 流（stream）概念 ：用来连接设备和过滤器，让数据得以流动；
- 其他概念 ：包括设备链（chain）、管道（pipe）、视图（view）等流处理相关的高级概念；
- 基于设备、过滤器等概念预定义的若干工具类：方便库用户的使用和自定义；
- 基本的流与流缓冲（stream 和 stream_buffer）：搭配设备使用，实现类似标准输入输出流的功能；
- 过滤流与过滤流缓冲（filtering_stream 和 filtering_streambuf）：增强的流，里面含有多个设备组成的链，数据流过链上的设备完成过滤处理；
- 大量的流操作函数 ：功能与标准库相近，但更加强大，其中最重要的是 copy()。

iostreams 库基本上是一个由头文件组成的库，位于目录<boost/iostreams/>下，大部分功能不需要编译就可以使用，头文件组织结构如下：

- <boost/iostreams/> ：库的核心功能，如流、概念实现、操作函数等；
- <boost/iostreams/device/>：预定义的设备类；
- <boost/iostreams/filter/>：预定义的过滤器类；
- <boost/iostreams/detail/>：库的实现细节，库用户通常无须关心。

iostreams 库需要编译后才能使用的功能包括 zlib/gzip/bzip2 格式的压缩解压缩、内存映射文件和操作系统的文件描述符，这些 cpp 源代码位于目录 <libs/iostreams/src/>下。因为这些源码没有复杂的编译依赖，因此需要时可直接加入工程编译（当然压缩功能还需要 zlib 等第三方库的支持）。如果需要使用正则表达式来处理字符流，那么还需要编译 boost.regex 库。如果嫌使用 bjam 编译麻烦，也可以集中在一个 cpp 文件中完成所有的编译，代码如下（需保证有 zlib、bzip 的库及头文件）：

```
// ioprebuild.cpp [2011/2/16 luojianfeng]
#define BOOST_IOSTREAMS_SOURCE        //直接使用 iostreams 库的源码

#include <libs/iostreams/src/zlib.cpp>
#include <libs/iostreams/src/gzip.cpp>
#include <libs/iostreams/src/bzip2.cpp>
#include <libs/iostreams/src/mapped_file.cpp>
#include <libs/iostreams/src/file_descriptor.cpp>
```

本章接下来的内容可分为两大部分，8.2 节～8.7 节我们将先研究 iostreams 库的基本使用方法，8.7 节之后再研究更进一步的定制功能。

为方便书写代码，本章假定有如下的名字空间定义：

```
namespace io = boost::iostreams;     //iostreams 名字空间别名
using namespace io;                  //所有 iostreams 功能均位于此名字空间
```

8.2 入门示例

本节我们将概览 iostreams 库的既有类和函数，初步了解流处理的各种基本概念和使用方法。

我们从两个示例程序开始，它们演示了 iostreams 一些组件的基本用法，可以让我们对流处理有个初步的了解。

8.2.1 示例 1

第一个示例程序比较简单，使用了类似标准流的功能：

```
#include <boost/iostreams/stream.hpp>         //基本流处理所需的头文件
#include <boost/iostreams/device/array.hpp>   //预定义的数组设备头文件
```

```
namespace io = boost::iostreams;                //iostreams 名字空间别名
using namespace io;                             //打开名字空间

int main()
{
    char str[] = "123";                         //字符数组
    array_source asrc(str, str + 3);            //一个数组源设备，注意构造函数
    stream<array_source> in(asrc);              //搭配源设备定义输入流

    char c1, c2, c3;
    in >> c1 >> c2;                             //使用读取操作符从输入流读取数据
    assert(c1 == '1' && c2 == '2');

    in.get(c3);                                 //可以调用标准流的成员函数
    assert(c3 == '3' && in.good());
    assert(in.get(c3).eof());                   //流已经结束

    char str2[10];                              //另一个字符数组
    array_sink asnk(str2);                      //定义接收设备，支持直接传入数组
    stream<array_sink> out(asnk);               //搭配接收设备定义输出流

    out << 'a' << 'b' << 'c';                   //使用写入操作符向流写入数据
    assert(str2[0] == 'a' && str2[2] == 'c');
}
```

这段代码中用到了三个 iostreams 库的组件：array_source、array_sink 和 stream，用法看起来非常像标准流 cin/cout。

array_source 是 iostreams 库提供的一个设备（参见 8.4.2 小节），它可以把一个字符缓冲区（字符数组）适配成一个源设备。有了源设备，stream 就可以用模板参数+构造函数的方式连接到这个设备，形成一个与标准流完全兼容的新的输入流。注意，因为流连接的设备是源设备，而源设备是可读不可写的，所以新流必然是一个输入（只读）流。接下来我们就可以如同标准输入流 cin 一样使用 operator>>或者 get()函数从流中也就是字符缓冲区中读取数据，当流被耗尽无数据可读时可以用成员函数 eof()来检测。

同样的，array_sink 是 iostreams 库提供的另一个设备，它可以把一个字符缓冲区适配成一个接收设备，stream 连接 array_sink 后就成为了一个输出流，可以写入数据，对流的写入操作最终都会被字符缓冲区接收。

8.2.2 示例 2

下面我们再来看一个稍微复杂一些的例子，它演示了过滤流、管道和 iostreams 库中最重要的 io::copy()算法的用法：

```
#include <boost/iostreams/stream.hpp>             //基本流处理所需的头文件
#include <boost/iostreams/device/array.hpp>       //预定义的数组设备头文件
#include <boost/iostreams/filtering_stream.hpp>   //过滤流头文件
#include <boost/iostreams/device/back_inserter.hpp> //接收设备适配头文件
#include <boost/iostreams/copy.hpp>               //io::copy 算法头文件
#include <boost/iostreams/filter/counter.hpp>     //计数过滤器头文件
namespace io = boost::iostreams;                  //iostreams 名字空间别名
using namespace io;                               //打开名字空间

int main()
{
  char arr[] = "12345678";                        //字符数组
  stream<array_source> in(arr, arr + 8);          //直接创建输入流

  string str;                               //标准字符串类，也可以算是标准容器
  filtering_ostream out(                    //输出过滤流，需连接 sink 设备作为链结束
      counter() |                           //预定义的计数过滤器
      io::back_inserter(str));              //适配标准容器作为接收设备

  io::copy(in, out);                        //调用 io::copy()算法
  assert(str.size() == 8);
  assert(str == arr);
  assert(out.component<counter>(0)->characters() == 8);  //获得计数结果
}
```

这段代码中没有用源设备，而是直接把字符缓冲区传递给了 stream 的构造函数创建了一个长度为 8 的输入流，写法更加简单，详细的原因可参见 8.6.1 小节。

接下来的过滤流是代码的核心功能所在：filtering_ostream 声明了一个用于输出（可写）的过滤流 out，与基本流不同的地方是它不仅可以连接接收设备，也可以连接过滤器，更可以把这些设备连成一个处理链。过滤流 out 的构造函数中传入了两个设备对象，中间用重载的 operator“|”连接。“|”被称为管道操作符，这与 Unix 中的管道操作符的概念非常相似，数据可以从一个设备通过管道流向另一个设备，更详细的信息可参见 8.5.2 小节。

过滤流设备链中的第一个设备是 counter（参见 8.5.3 小节），它是 iostreams 库预定义的一个过滤器，可以计算通过过滤器的字符数和行数。第二个设备使用了 io::back_inserter()函数（参见 8.4.3 小节），它是一个接收设备生成器，可以把一个标准容器适配成一个接收设备。

最后我们调用 io::copy()算法，它与标准库的 copy()算法非常相似（参见 8.7 小节）。标准库的 copy()操纵迭代器，把数据从一个可读迭代器拷贝到另一个可写迭代器，而 io::copy()则用来操作流，把数据从一个输入流拷贝到输出流。这样，数据就从输入流开始，先流过 counter 过滤器，然后再流入被适配的 string 容器，完成了流处理过程。[①]

代码的最后一行调用了过滤流的成员函数 component<>()，它可以返回流的设备链中的第 n 个设备，在这里就是第一个过滤器 counter。然后再调用 counter 的成员函数 characters()获得字符数统计。

8.3　设备的特征

通过上一节的两个代码示例，我们初步了解了 iostreams 的工作机制和 iostreams 中的源设备、接收设备、过滤器和流的用法，然而，要让这些设备一起协同工作，它们必须要满足一定的要求，也就是我们在元编程领域中经常提到的 traits。

头文件<boost/iostreams/traits.hpp>中定义了 iostreams 库的所有 traits 相关工具。

8.3.1　设备的字符类型

设备最基本的特征是它们能够处理的“字符类型”（char_type），流上的所有设备协同工作的最低要求是它们必须处理的是同一种“字符类型”。

设备的“字符类型”可以使用元函数 char_type_of<T>::type 获得，它类似于标准库的 std::char_traits<Ch>::char_type：

```
#include <boost/type_traits.hpp>
#include <boost/iostreams/traits.hpp>

int main()
{
```

① 流处理与迭代器处理有很多相似性，进一步的讨论参见 8.11 小节。

```
    //数组设备的字符类型是 char
    assert((is_same<char_type_of<array_source>::type,
                    char>::value));
    assert((is_same<char_type_of<array_sink>::type,
                    char>::value));

    //以 w 开头的是同名设备的宽字符版本
    assert((is_same<char_type_of<warray_source>::type,
                    wchar_t>::value));
    assert((!is_same<char_type_of<warray_sink>::type,
                    char_type_of<array_sink>::type>::value));
}
```

注意：我们所说的“字符类型”，不一定必须是 char 或者 wchar_t——虽然大多数情况下流都是处理真正的字符。但有的时候我们也可以处理其他的数据类型，例如 unsigned char 或者 int 等，这个时候整个流上的设备的“字符类型”必须是一致的，否则会导致编译失败。例如，如果有一个处理 char 的源设备，那么随后的过滤器和接收设备也必须是 char 设备，如果使用 wchar_t 的设备（如 wsink）就会无法通过编译。

8.3.2 设备的模式

设备的另一个重要特征是它的模式（mode）。在 iostreams 库中，模式指的是设备的输入输出（读写）访问特征，与 C 语言的文件模式或新式迭代器的遍历概念比较接近（参见 3.1.3 小节）。

最常用的模式有四种：

- 输入模式（input）：可以在一个字符序列上执行读操作，如 std::cin。
- 输出模式（output）：可以在一个字符序列上执行写操作，如 std::cout。
- 双向模式（bidirectional）：可以在两个字符序列上分别执行读写操作，如 std::iostream。
- 可定位模式（seekable）：可以在一个字符序列上执行读写操作，但可重定位操作位置，如 std::fstream。

这四个模式的关系图如图 8-2 所示（摘自 Boost 文档）：

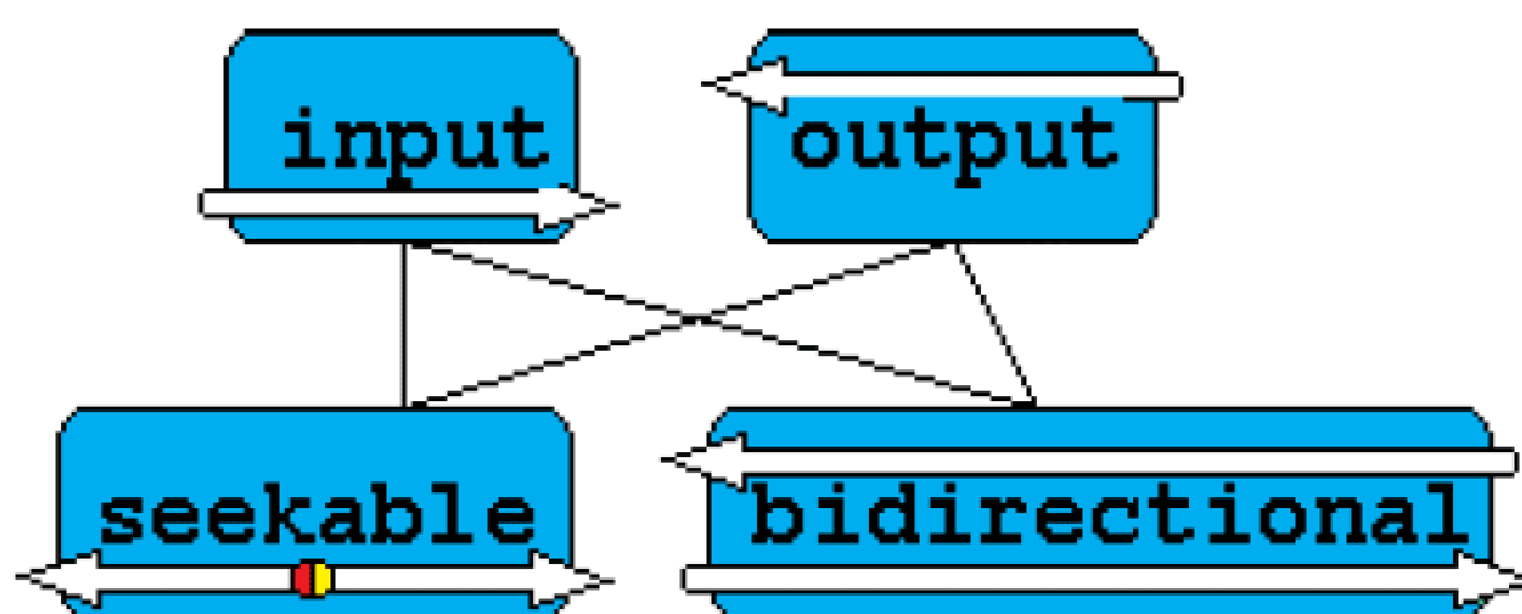


图 8-2 四模式关系图

另外还有四个模式，它们也是 iostreams 库的组成部分，但用途没有上面的四个那么广泛：

- 可定位输入模式（input_seekable）：可以在一个字符序列上执行读操作，可重定位操作位置，如 std::ifstream；
- 可定位输出模式（output_seekable）：可以在一个字符序列上执行写操作，可重定位操作位置，如 std::ofstream；
- 双可定位模式（dual_seekable）：可以在一个字符序列上执行读写操作，读写操作可独立地重定位位置，如 std::stringstream；
- 双向可定位模式（bidirectional_seekable）：可以在两个字符序列上执行读写操作，可重定位操作位置。

全部八个模式的关系图如图 8-3 所示（摘自 Boost 文档）：

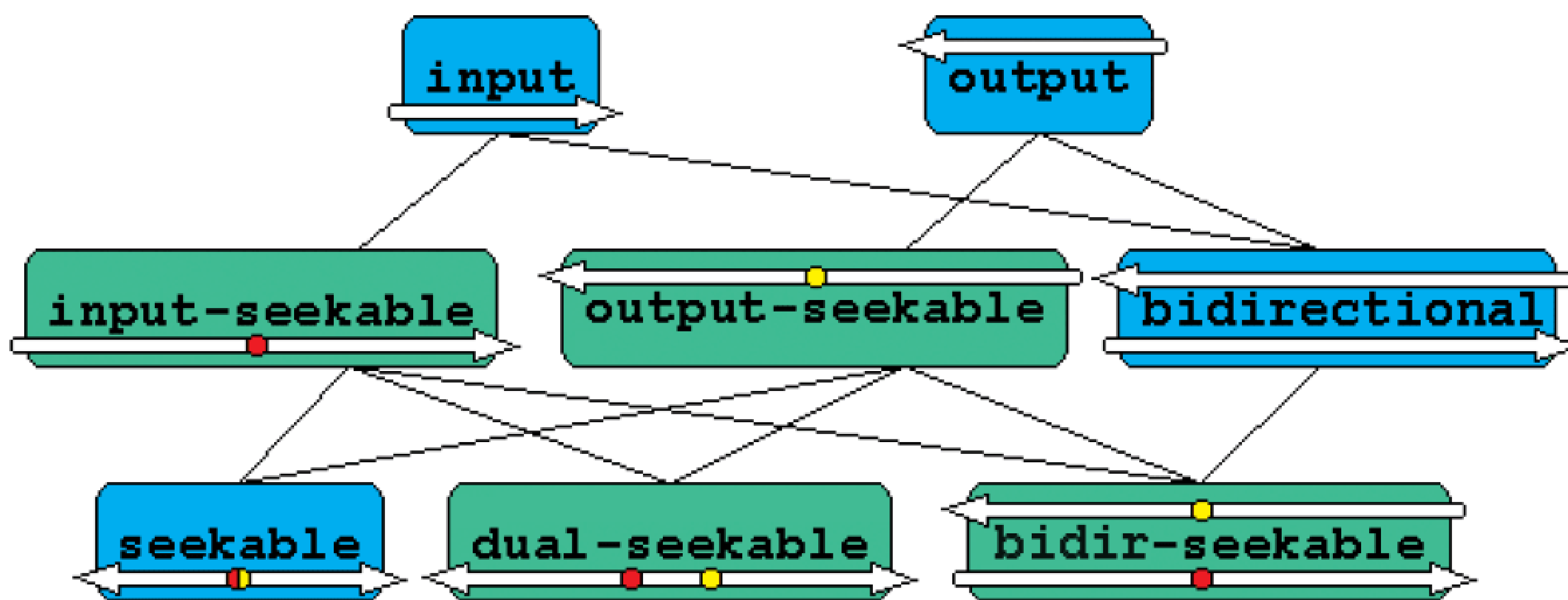


图 8-3 八模式关系图

使用元函数 mode_of<T>可获得设备的模式，例如：

```
//source 是 iostreams 库中的一个源设备基类
```

```
assert((is_same<mode_of<source>::type,
                 input>::value));

//检查 array_source 和 array_sink 的模式
assert((is_same<mode_of<array_source>::type,
                 input_seekable>::value));
assert((is_same<mode_of<array_sink>::type,
                 output_seekable>::value));

//array 是一个可读可写的设备
assert((is_same<mode_of<array>::type,
                 seekable>::value));
```

除了这八种模式，为了方便起见 iostreams 库还定义了一个“伪模式”dual_use，表示设备即可以用于输入也可以用于输出，但不能同时用于输入输出，所以也可以把它称为单向模式。

8.3.3　设备的分类

字符类型和访问模式是设备的两个最重要的特征，此外 iostreams 还定义了许多其他的特征来更精确地描述设备，例如设备的类别（源设备、接收设备还是过滤器）、是否可关闭、是否可阻塞等，这些特征 tag 类都位于头文件<boost/iostreams/categories.hpp>[①]。

设备的分类（category）是除字符类型外所有设备特征的总和，它表现为设备的一个内部类型 category，使用多重继承的方式派生自各种 traits 类：

```
struct category                            //一个可关闭的输入设备分类定义
    : input, device_tag, closable_tag      //包含了多个特征
    { };
```

因此 category 可以转型到其他特征 tag。

元函数 category_of<T>可以获得设备的分类信息，另有一个自由函数 get_category (const T&)包装了 category_of<T>，可以直接返回分类对象。示例代码如下：

```
assert((is_convertible<category_of<array_source>::type,
                input_seekable>::value));
assert((is_convertible<category_of<array_source>::type,
```

① categories.hpp 里定义了大量用做标志的 tag 类，本书不能一一介绍，读者可阅读源代码进行研究。

```
                device_tag>::value));

assert((is_convertible<category_of<counter>::type,
                filter_tag>::value));
assert((is_convertible<category_of<counter>::type,
                dual_use>::value));
```

iostreams 库还提供其他一些获取设备属性的元函数，我们将在后面逐步介绍。

8.4 设备

设备是 iostreams 库中的一个重要概念，本节我们首先研究基本概念，然后学习四个常用的预定义设备。

判断一个类是否是设备可以使用元函数 is_device<T>。

8.4.1 设备概述

设备(device)是iostreams库中最基础的概念,它位于头文件<boost/iostreams/concepts.hpp>，是一个可以对序列执行读或写操作的对象，其行为取决于它的模式（见上节）。

device 是一个模板类，类摘要如下：

```
template<typename Mode, typename Ch = char>
struct device {
    typedef Ch char_type;       //字符类型
    struct category             //设备的分类，有缺省值
        : Mode, device_tag, closable_tag, localizable_tag{ };

    void close();
    void close(openmode);
};
```

device 是一个只被用于继承的概念类，它有两个模板参数，参数 Mode 决定了设备的模式，参数 Ch 是设备能够处理的字符类型，默认是窄字符 char。

device 有两个非常重要的内部类型定义，分别是 char_type 和 category，它们标记了设备的字符类型和分类信息，可以用相应的元函数 char_type_of<>和 category_of<>获得。

为了方便用户使用，device 还定义了若干特化，它们分别表示一些设备子概念[①]，例如：

```
typedef device<input>    source;       //char 源设备
typedef device<output>   sink;         //char 接收设备
```

iostreams 提供了数个预定义的设备，下面我们着重介绍数组设备、标准容器设备（适配器）、文件设备和空设备，其他设备读者可自行研究。

8.4.2 数组设备

数组设备位于头文件<boost/iostreams/device/array.hpp>，它提供了对内存字符序列的访问，可以把字符缓冲区适配成 seekable 设备。数组设备不负责缓冲区的内存管理，因此它通常搭配静态数组或者 std::vector 来使用。

iostreams 库提供了三个数组设备，分别是 array_source、array_sink 和 array，分别是源设备、接收设备和可定位设备，我们之前已经初步见到了它们的使用方法。

类摘要

这三个数组设备实际上是basic_array_source、basic_array_sink和basic_array 的 char 类型特化，它们仅在模式上不同，最后的实现都是同一个类 array_adapter。

array_adapter 的类摘要如下：

```
template<typename Mode, typename Ch>
class array_adapter {
public:
    typedef Ch                                  char_type;  //字符类型
    typedef std::pair<char_type*, char_type*>  pair_type;
    struct category                                         //设备的分类
        : public Mode, public device_tag, public direct_tag
        { };
    array_adapter(char_type* begin, char_type* end);
    array_adapter(char_type* begin, std::size_t length);
    array_adapter(char_type (&ar)[N])
        : begin_(ar), end_(ar + N) { }

    pair_type input_sequence();
```

① 这些子概念均有窄字符和宽字符对应的版本，为叙述简单起见本书忽略宽字符版本，如 wsource、wsink 等，请读者了解。

```
    pair_type output_sequence();
private:
    char_type* begin_;
    char_type* end_;
};
```

解说

array_adapter 符合设备的概念，因此它的模板参数、字符类型和分类的含义均与 device 相同。

array_adapter 的构造函数是它的主要功能，能够用以下三种形式把一个数组适配成设备：

- 两个首尾指针（可为 const）标明数组的起点和终点；
- 数组起点（可为 const）和数组的长度；
- 直接使用数组的引用形式，数组的长度使用模板推导。

另外两个成员函数 input_sequence()和 output_sequence()返回一个 pair 对象，标记了数组设备所能控制的序列范围，即成员变量 begin_和 end_。

用法

数组设备的用法我们之前已经看过了，下面再简单地举几个例子，重点是演示其构造函数：

```
char buf[] = "123";              //字符数组
array_source as1(buf, buf + 3);  //数组设备 1
array_source as2(buf);           //数组设备 2

assert(as1.input_sequence().first == buf);
assert(as1.input_sequence().second == buf + 3);
assert(as2.input_sequence().second == buf + 4);
```

请读者注意代码中两个数组设备的声明形式，as1 使用的是首末指针的方式，因此流的读取长度是 3，而 as2 使用的是数组引用方式，因此流的读取长度是数组的真正长度 4（含字符串的最后一个 null 结束标记）。

我们也可以改变字符类型定制操作其他数据类型的设备，这时就不能直接使用 array_source、array_sink 这些处理 char 的设备，而是要使用 basic_array_source

等模板类自行特化，例如：

```
int buf1[10] = {1, 2, 3}, buf2[10];                 //两个整型数组
basic_array_source<int> ai(buf1);                   //使用 int 类型的源设备
basic_array<int> ar(buf2);                          //使用 int 类型的可定位设备

io::copy(                                           //调用 copy 算法处理流
      stream<basic_array_source<int> >(ai),
      stream<basic_array<int> >(ar));
```

8.4.3 标准容器设备

如果仅能使用原始数组，而不能使用标准容器用来输入输出那可实在是太不方便了，iostreams 库当然考虑到了这方便的需求，对标准容器提供了很好的支持。

源设备

iostreams 库不能把一个标准容器直接适配成源设备，但借助 boost.range 库提供的 boost::make_iterator_range()函数，可以把容器直接传递给过滤流创建出可用于输入的流。示例代码如下：

```
#include <boost/iostreams/filtering_stream.hpp>
int main()
{
  string str("123");                                    //标准字符串容器
  filtering_istream in(make_iterator_range(str));       //创建输入流

  vector<char> v(str.begin(), str.end());               //标准向量容器
  filtering_istream in2(make_iterator_range(v));        //创建输入流
}
```

接收设备

iostreams 库为标准容器提供了一个接收设备的适配类 back_insert_device，它位于头文件<boost/iostreams/device/back_inserter.hpp>，类摘要如下：

```
template<typename Container>
class back_insert_device {
public:
    typedef typename Container::value_type  char_type;     //字符类型
    typedef sink_tag                         category;     //设备的分类
```

```
        back_insert_device(Container& cnt);                        //构造函数
    protected:
        Container* container;
    };
```

back_insert_device 调用了标准容器的 insert()操作，向容器的末尾追加数据，与 std::back_insert_iterator 很像但更好。

back_insert_device 可以这样使用：

```
string str("123");
back_insert_device<std::string> bid(str);              //适配成输出设备
stream<back_insert_device<std::string> > out(bid);     //输出流
//也可直接写成 stream<back_insert_device<std::string> > out(str);
```

为了方便使用，iostreams 库提供工厂函数 back_inserter(Container& cnt)，它可以直接返回一个被适配的接收设备，这在使用过滤流或者 io::copy 算法时非常有用：

```
int main()
{
    string str("123");      //字符串标准容器
    vector<char> v();       //标准向量容器

    io::copy(               //调用 io::copy 算法，使用过滤流
        filtering_istream(make_iterator_range(str)),
        filtering_ostream(io::back_inserter(v)));
};
```

容器的设备适配器

使用上述的两种方法可以把标准容器用于流处理，但无论如何，我们还是没有符合设备概念的容器设备，有的时候可能真的非常需要（虽然可能性很小）。

iostreams 库在示例代码<libs/iostreams/example/container_device.hpp>中提供了三个可用的模板类：container_source、container_sink 和 container_device，它们可以把符合随机访问遍历概念的标准容器（包括 vector、string、deque）适配成设备。

如果确有需要，读者可以使用这三个适配器类，注意它们位于名字空间 boost::iostreams::example。当然，也希望有那么一天 iostreams 能把它们“扶正”成为库的正式组件。

8.4.4 文件设备

文件设备位于头文件<boost/iostreams/device/file.hpp>，它们基于标准库的文件流提供了对文件的访问，用法与标准库的 fstream 颇类似。

iostreams 库提供了三个文件设备，分别是 file_source、file_sink 和 file，分别是源设备、接收设备和可定位设备。

类摘要

这三个文件设备实际上是 basic_file_source、basic_file_sink 和 basic_file 的 char 类型特化，核心类是 basic_file。

basic_file 的类摘要如下：

```
template<typename Ch>
class basic_file {
public:
    typedef Ch char_type;       //字符类型
    struct category             //设备的分类
        : public seekable_device_tag,
          public closable_tag,
          public localizable_tag
        { };
    basic_file( const std::string& path,
                openmode mode);
    std::streamsize read(char_type* s, std::streamsize n);
    std::streamsize write(const char_type* s, std::streamsize n);
    bool putback(char_type c);
    std::streampos seek( stream_offset off, seekdir way,
                         openmode which = in | out);
    void open( const std::string& path, openmode mode);
    bool is_open() const;
    void close();
private:
    struct impl {               //内部实现类
        impl(const std::string& path, openmode mode)
            { file_.open(path.c_str(), mode); }
```

```
        ~impl() { if (file_.is_open()) file_.close(); }
        std::basic_filebuf file_;
    };
    shared_ptr<impl> pimpl_;
};
```

解说

basic_file 使用 shared_ptr 和 pimp 惯用法包装了 std::basic_filebuf，因此用户无须关心文件的打开关闭问题，shared_ptr 可以自动管理生命周期。

basic_file 的用法与 std::fstream 非常相似，可以使用文件名构造直接打开文件，使用 read() 和 write() 读写数据，seek() 移动文件指针的位置。

用法

示范文件设备流用法的代码如下：

```
#include <boost/iostreams/device/file.hpp>
int main()
{
  string str("file device");            //标准字符串
  file_sink fsink("test.txt");          //文件接收设备

  io::copy(                             //io::copy 算法把字符串写入文件
      make_iterator_range(str),         //可以直接使用 make_iterator_range()
      stream<file_sink>(fsink));        //文件接收设备接收

  file_source fsrc("test.txt");         //文件源设备
  io::copy(                             //从文件流中读取字符，拷贝到标准输出流
      stream<file_source>(fsrc),
      cout);
}
```

8.4.5 空设备

空设备位于头文件<boost/iostreams/device/null.hpp>，它们是空对象模式的具体应用，是不执行任何动作的设备。

iostreams 库提供了两个空设备，分别是 null_source 和 null_sink，分别是源设备和接收设备。

类摘要

null_source 和 null_sink 实际上是 basic_null_source 和 basic_null_sink 的 char 类型特化，它们仅是输入输出模式不同，最后的实现都是同一个类 basic_null_device。

basic_null_device 的类摘要如下：

```
template<typename Ch, typename Mode>
class basic_null_device {
public:
    typedef Ch char_type;      //字符类型
    struct category            //设备的分类
        : public Mode,
          public device_tag,
          public closable_tag
        { };
    std::streamsize read(Ch*, std::streamsize) { return 0; }
    std::streamsize write(const Ch*, std::streamsize n) { return n; }
    std::streampos seek(...)
    { return -1; }
    void close() { }
    void close(BOOST_IOS::openmode) { }
};
```

解说

空设备的所有成员函数都不执行任何操作，不处理数据：用户无法从 null_source 中获得任何数据，因为 read() 函数总返回 0 字节的数据；用户可以向 null_sink 写入任意数据，但数据会被完全忽略，就像是写进了一个“黑洞”。

空设备可以用在某些特定的场合，比如完全不关心数据的来源或者去向，null_sink 的示例代码参见 8.5.3 小节。

8.5　过滤器

设备的概念很重要，它定义了数据的起点和终点，我们可以从源设备中读取数据，向接收设备写入数据，但仅有设备还不够，因为单纯的数据读写是没有意义的，更重要的是数据

在流转过程中的处理，这正是过滤器做的工作。

判断一个类是否是过滤器可以使用元函数 is_filter<T>。

8.5.1 过滤器概述

过滤器（filter）是一种特殊的设备，它不具有源设备或接收设备的读写能力，只能允许数据流过，但它能够对流过的数据执行任意操作，完成某些特殊的功能。

filter 是过滤器的概念类，它位于头文件<boost/iostreams/concepts.hpp>，类摘要如下：

```
template<typename Mode, typename Ch = char>
struct filter {
    typedef Ch char_type;                          //字符类型
    struct category                                //设备的分类
        : Mode, filter_tag, closable_tag, localizable_tag{ };

    template<typename Device>  void close(Device&);
    template<typename Device>  void close(Device&, openmode);
};
```

同 device 一样，filter 也有模式和字符类型两个模板参数，它们定义了 filter 的特征，也可以使用元函数 char_type_of<T>和 mode_of<T>等获取。

为了方便用户使用，filter 定义了若干特化用于表示子概念，例如：

```
typedef filter<input>      input_filter;      //输入过滤器
typedef filter<output>     output_filter;     //输出过滤器
typedef filter<seekable>   seekable_filter;   //可定位过滤器
typedef filter<dual_use>   dual_use_filter;   //两用过滤器
```

iostreams 库提供了大量的预定义的过滤器和辅助用户定制的过滤器，下面先介绍管道和设备链概念，然后着重介绍几个常用的过滤器。

8.5.2 管道和设备链

iostreams 库借鉴了 Unix 中的“管道”概念和形式，数据通过管道可以从一个设备流向另一个设备。因为重载了 operator|，所以我们可以使用“|”把多个过滤器连接成链，最后通常以一个设备结束，这个设备就是数据的起点或终点。

设备链通常的形式如下：

```
filter1 | filter2 | ... | filterN | filter-or-device
```

对于输出链，数据从左到右依次流过每一个过滤器，每一次流过过滤器时都被该过滤器处理，最后到达链的终点时写入接收设备。

对于输入链，数据的流向则刚好相反，数据先从最右端的源设备流入，然后从右到左依次流过每一个过滤器。

如果链的终点不是一个设备，那么链就称为“不完整的链”（incomplete chain），不能用于 IO 操作，反之则称为“完整的链”（complete chain）。

类摘要

chain 位于头文件<boost/iostreams/chain.hpp>，实际上是 chain_base 的子类，因为真实接口较复杂，故简化摘要如下：

```
template< typename Mode, typename Ch   = char >
class chain  {
public:
    typedef Ch                    char_type;          //字符类型
    typedef Mode                  mode;               //设备的模式
    typedef Tr                    traits_type;        //其他特征

    chain();
    chain(const chain&);

    std::streamsize read(char_type* s, std::streamsize n);
    std::streamsize write(const char_type* s, std::streamsize n);
    stream_offset seek(stream_offset off, std::ios_base::seekdir way);

    const std::type_info& component_type(int n) const;
    template<typename T>
    T* component(int n) const;

    template<typename T>
    void push( const T& t);
    template<typename StreamOrStreambuf>
    void push( StreamOrStreambuf& t);
```

```
    void pop();
    bool empty() const;
    size_type size() const;
    void reset();
    bool is_complete() const;
};
```

解说

chain 的主要功能就是管理设备链，因此它内部使用 std::list 来保存所有设备的拷贝，成员函数 component_type()可返回第 n 个设备的类型信息，而 component<>()则以指针的形式返回该设备。注意，component<>()的模板参数不能自动推导，必须手工指定，我们已经在 8.2.2 小节见过这样的用法。

chain 可以一开始构造为一个空链，也可以传入一个设备或者用管道连接的多个设备。如果链是空的或者不完整的，那么还可以随时使用成员函数 push()向链的末尾追加设备或流，pop()函数的功能则刚好相反——它从链的末尾删去一个设备。使用 push()的时候要注意，chain 存储设备的拷贝，因此有时候可能需要 boost.ref 库的帮助，它可以包装引用。

chain 还提供一些简单的成员函数用来查看链的属性，如判断是否空、是否完整，获得链的长度等等，都比较简单，读者可自行参考文档了解。

管道

iostreams 库的管道功能位于头文件<boost/iostreams/pipeline.hpp>，提供了一个辅助类 pipeline 和重载的 operator|操作符，使我们可以用“语法糖”的形式简单地把设备 push 到链中。

但一个不幸的消息是，被 boost.ref 库包装后的引用不能够使用管道功能，这是因为经过 ref 库包装后返回的是 reference_wrapper 类型，没有为它定义的 operator“|”重载。

8.5.3 计数过滤器

计数过滤器位于头文件<boost/iostreams/filter/counter.hpp>，是一个两用（dual_use）过滤器，也就是说既可以用做输入也可以用做输出。它是一个“透明”的过滤器，不对流经的数据做任何更改，仅仅统计字符数和行数。

类摘要

basic_counter 是计数过滤器的基本类，counter 只是 basic_counter 的 char 特化。

basic_counter 的类摘要如下：

```
template<typename Ch>
class basic_counter  {
public:
    typedef Ch char_type; //字符类型
    struct category       //设备的分类
        : dual_use, filter_tag,
          multichar_tag, optimally_buffered_tag
        { };
    explicit basic_counter(int first_line = 0, int first_char = 0);
    int characters() const { return chars_; }
    int lines() const { return lines_; }
private:
    int lines_;             //行数
    int chars_;             //字符数
};
```

用法

counter 是一个很简单的过滤器，它的用法也很简单，只需要把它加入设备链，用 io::copy 算法让数据流经它，就可以用 characters()和 lines()获得统计数据了。

我们已经在 8.2.2 小节中看到了 counter 的使用例子，下面再用代码简单地示范一下：

```
#include <boost/iostreams/device/null.hpp>
#include <boost/iostreams/filter/counter.hpp>
#include <boost/iostreams/filtering_stream.hpp>

int main()
{
  string str("counter\nfilter\n");              //标准字符串
  //过滤流，使用计数过滤器和空接收设备构成链
  filtering_ostream out(counter() | null_sink() );

  io::copy(                                     //io::copy 算法流处理
```

```
        boost::make_iterator_range(str),
        out );

    counter* pc = out.component<counter>(0);            //获得计数过滤器指针
    assert(pc->characters() == 15);
    assert(pc->lines() == 2);
}
```

因为只统计字符数，所以这段代码中使用了 null_sink 这个空设备，它只是简单地消耗输入，不做任何输出动作。使用 io::copy 算法后调用函数 component<>()获得计数过滤器的指针，然后得到统计数据。

8.5.4 正则表达式过滤器（Ⅰ）

iostreams 库提供两个正则表达式过滤器，第一个过滤器的名字是 regex_filter，它在流里搜索匹配的正则表达式，匹配成功则替换文本，位于头文件<boost/iostreams/filter/regex.hpp>。

由于使用了 boost.regex 库，如果要使用 regex_filter 必须先编译它①。

类摘要

basic_regex_filter 是正则表达式过滤器的核心类，摘要如下：

```
template< typename Ch>
class basic_regex_filter
    : public aggregate_filter<Ch, ...>
{
public:
    typedef typename base_type::char_type  char_type;     //字符类型
    typedef typename base_type::category   category;      //设备的分类
    typedef std::basic_string<Ch>          string_type;
    typedef basic_regex<Ch, Tr>            regex_type;
```

---

① regex 是 Boost 中的一个正则表达式解析库，需要编译后才能使用，除了 Boost 标准的 bjam 外它还提供了搭配各种流行 C++编译器的 Makefile，位于<libs/regex/build/>下，读者可以任意选择自己熟悉的方式构建 regex 库。

对于 VC8 来说，最简单的方式就是把 regex 的所有 cpp 文件加入到工程中以源代码的方式直接使用，同时在包含<boost/regex.hpp>前#define BOOST_REGEX_SOURCE 或#define BOOST_REGEX_NO_LIB 禁用 VC 的动态链接功能。

因为正则表达式已经在推荐书目[1]的 xpressive 库做过详细介绍，故本书不对 regex 库再重复。

```
    typedef match_results<const Ch*>      match_type;

    typedef function1<string_type, const match_type&>  formatter;

    basic_regex_filter( const regex_type & pattern,
                        const string_type & fmt);
    basic_regex_filter( const regex_type & pattern,
                        const Ch* fmt);
    basic_regex_filter( const regex_type & pattern,
                        const formatter& replace);
};
```

为简单起见，类摘要中忽略了正则表达式的匹配标志参数。

解说

basic_regex_filter 使用 regex 库执行正则表达式匹配，因此在构造时必须传入一个 regex 对象。构造函数的第二个参数是替换的目标文本，可以直接使用 C 字符串或者标准字符串形式的简单文本。

需要留意 basic_regex_filter 第三种形式的构造函数，它使用了 boost.function 库，可以接受任意一个用于格式化的单参函数或函数对象。这个函数接受正则表达式匹配结果的 match_results，然后返回处理后的字符串。因为有了这个格式化函数的间接层，就可以在里面利用正则表达式的全部功能做任何事情。

用法

regex_filter 的用法也很简单，与 regex 库的 regex_replace()的功能基本相同，但手法却是流处理的形式。

第一个例子使用了简单的文本替换，把表达式“a.c”替换为“test”:

```
#define BOOST_REGEX_SOURCE
#include <boost/iostreams/filter/regex.hpp>
#include <boost/iostreams/copy.hpp>
#include <boost/iostreams/filtering_stream.hpp>
#include <boost/iostreams/device/back_inserter.hpp>
int main()
{
  regex reg("a.c");                          //构造一个正则表达式
  string str("abcdef aochijk");              //要被处理的字符串
```

```
    string result;                          //结果字符串

    io::copy(                               //io::copy 算法流处理
        boost::make_iterator_range(str),    //适配字符串为输入设备
        filtering_ostream(                  //在算法中直接创建过滤流
            regex_filter(reg, "test") |     //正则表达式过滤器
            io::back_inserter(result))      //标准容器接收设备
        );
    cout << result << endl;                 //输出替换结果
}
```

程序的运行结果如下：

```
testdef testhijk
```

接下来我们编写一个处理 match_results 的格式化函数，用来处理正则表达式的匹配结果：

```
string reg_format(const match_results<const char*>& match)
{
    return string("test-") + match[1] + "-";//使用第 1 个子表达式
}
```

为了使用格式化函数正则表达式必须做出一点改动，增加子表达式的定义，全部代码如下：

```
int main()
{
    regex reg("a(.)c");                     //使用圆括号定义了一个子表达式
    string str("abcdef aochijk");
    string result;

    io::copy(                               //io::copy 算法流处理
        boost::make_iterator_range(str),    //适配字符串为输入设备
        filtering_ostream(                  //在算法中直接创建过滤流
            regex_filter(reg, reg_format)|  //使用格式化函数
            io::back_inserter(result))      //标准容器接收设备
        );
    cout << result << endl;                 //输出替换结果
}
```

程序的运行结果如下：

```
test-b-def test-o-hijk
```

8.5.5 正则表达式过滤器（II）

grep_filter 是 iostreams 库里的另一个正则表达式过滤器，提供类似 Unix 工具 grep 的功能，可以抓取流中含有匹配的行，它位于头文件<boost/iostreams/filter/grep.hpp>

类摘要

basic_grep_filter 是 grep_filter 的核心类，摘要如下：

```
namespace grep {                                           //grep 选项定义
const int invert     = 1;                                  //保留不匹配的行
const int whole_line  = invert << 1;                       //保留匹配的行
} // End namespace grep.

template< typename Ch>
class basic_grep_filter : public basic_line_filter<Ch> {
public:
    typedef typename base_type::char_type  char_type;      //字符类型
    typedef typename base_type::category   category;       //设备的分类

    basic_grep_filter( const regex_type& re,               //正则表达式
                       match_flag_type match_flags ,       //正则表达式匹配标志
                       int options = 0 );                  //grep 选项
    int count() const;
};
```

用法

grep_filter 的主要功能集中在它的构造函数，它使用正则表达式匹配流中的字符串，并根据 options 来决定是匹配 grep 还是不匹配 grep。options 的取值定义在子名字空间 boost::iostreams::grep 里，如果是 whole_line，那么保留匹配的行；如果是 invert，那么保留不匹配的行。最后提取的行数可以使用成员函数 count()获得。

下面的代码与 regex_filter 很接近，用来提取含有正则表达式的行：

```
#define BOOST_REGEX_SOURCE
#include <boost/iostreams/filter/grep.hpp>
#include <boost/iostreams/copy.hpp>
#include <boost/iostreams/filtering_stream.hpp>
```

```
int main()
{
  regex reg("a.c");                       //构造一个正则表达式
  string str( "abcdef\n"                  //要被处理的多行字符串
              "12345\n"
              "aochijk\n");
  string result;                          //结果字符串

  io::copy(                               //io::copy 算法流处理
      boost::make_iterator_range(str),    //适配字符串为输入设备
      filtering_ostream(                  //在算法中直接创建过滤流
          grep_filter(reg) |              //正则表达式过滤器，缺省选项
          io::back_inserter(result))      //标准容器器接收设备
      );
  cout << result << endl;                 //输出提取结果
}
```

程序的运行结果是：

```
abcdef
aochijk
```

如果使用 invert 方式来提取行，代码为：

```
io::copy(
    boost::make_iterator_range(str),
    filtering_ostream(
        grep_filter(reg,
            regex_constants::match_default, grep::invert) |
        io::back_inserter(result))
    );
```

运行后将输出第二行“12345”。

8.5.6 压缩过滤器

iostreams 库提供了三个压缩过滤器，分别是 zlib_compressor/zlib_decompressor、gzip_compressor/gzip_decompressor 和 bzip2_compressor/bzip2_decompressor，它们都必须配合编译后的第三方库才能使用，读者可自行查找相应压缩算法库的编译方法。

因为这三个压缩过滤器用法类似，下面我们主要介绍 zlib_compressor/zlib_decompressor，它使用了 zlib 的压缩算法，位于头文件<boost/iostreams/filter/zlib.hpp>，需要编译源文件<libs/iostreams/src/zlib.cpp>。

类摘要

zlib_compressor/zlib_decompressor 实际上是 basic_zlib_compressor/basic_zlib_decompressor 的特化形式，前者用于压缩，而后者用于解压缩，这两个过滤器都是两用（dual_use）过滤器，因此既可以用于输入也可以用于输出。

basic_zlib_compressor 和 basic_zlib_decompressor 的类摘要如下：

```
template<typename Alloc = std::allocator<char> >
struct basic_zlib_compressor: symmetric_filter<>
{
public:
    basic_zlib_compressor( const zlib_params&, int buffer_size );
    zlib::ulong crc();
    int total_in();
};
template<typename Alloc = std::allocator<char> >
struct basic_zlib_decompressor: symmetric_filter<>
{
public:
    basic_zlib_decompressor( int window_bits, int buffer_size);
    basic_zlib_decompressor( const zlib_params& p, int buffer_size);
    zlib::ulong crc();
    int total_out();
};
```

basic_zlib_compressor 和 basic_zlib_decompressor 没有太多可说的，它们调用 zlib 库实现压缩解压缩功能，构造函数都有缺省值，不需要特别的配置就可以工作得很好。

子名字空间 boost::iostreams::zlib 里定义了 zlib 算法所需要的全部参数信息，可以设置压缩级别、压缩方法、压缩策略等参数，可以使用 zlib_params 结构传递给压缩过滤器。zlib_params 结构摘要如下：

```
struct zlib_params {
    zlib_params( int level          = zlib::default_compression,
```

```
                int method              = zlib::deflated,
                int window_bits         = zlib::default_window_bits,
                int mem_level           = zlib::default_mem_level,
                int strategy            = zlib::default_strategy,
                bool noheader           = zlib::default_noheader,
                bool calculate_crc      = zlib::default_crc );
};
```

由于参数较多，本书不做过多的介绍，有需要的读者请参考 Boost 文档或者源代码。

用法

看过了之前那么多过滤器和流的介绍，相信读者此时已经对过滤器的使用比较熟悉了，而且压缩过滤器也确实很容易使用(只要不去定制复杂的压缩策略)，下面直接给出示范代码：

```
#define BOOST_IOSTREAMS_SOURCE               //把源代码加入工程编译，不用动态链接
#include <boost/iostreams/filter/zlib.hpp>
#include <boost/iostreams/copy.hpp>
#include <boost/iostreams/filtering_stream.hpp>

int main()
{
  string str("12345678 12345678");          //待压缩的数据
  string zip_str, unzip_str;                //保存压缩和解压缩的数据

  io::copy(                                 //io::copy 算法流处理
      boost::make_iterator_range(str),      //适配字符串为输入设备
      filtering_ostream(                    //使用压缩过滤器输出
          zlib_compressor() | io::back_inserter(zip_str) )
      );

  io::copy(                                 //io::copy 算法流处理
      boost::make_iterator_range(zip_str),//适配字符串为输入设备
      filtering_ostream(                    //使用解压缩过滤器输出
          zlib_decompressor() | io::back_inserter(unzip_str) )
      );

  assert(unzip_str == str);                 //解压缩后数据还原
}
```

如果在压缩处理过程中发生错误，那么压缩过滤器会抛出异常 zlib_error，它是 std::exception 的子类，可以用成员函数 error() 获得与 zlib 兼容的错误代码。

8.6 流

设备和过滤器仅是一些孤立的对象，而流则是连接设备的纽带，只有通过流数据才能从一个设备向下一个设备转移，在流转的过程中实现数据的处理。

前几节我们已经多次使用了基本流 stream 和过滤流 filtering_ostream，本节将对它们做详细的讨论。

8.6.1 基本流

在 iostreams 库中基本流有 stream 和 stream_buffer 两个模板类，两者的功能和行为很相似，为简单起见，我们只讨论 stream，它位于头文件<boost/iostreams/stream.hpp>。

类摘要

stream 使用了模板元编程技术，可以根据模板参数的特征信息推导出它的父类，可能是 std::basic_istream、std::basic_ostream 或者 std::basic_iostream，简化的摘要如下：

```
template< typename T >
class stream  {
public:
    stream();
    stream( const T& t);
    template<typename U1, ..., typename UN>
    stream([const] U1& u1, const U2& u2, ..., const UN& uN);

    void open( const T& t);
    template<typename U1, ..., typename UN>
    void open([const] U1& u1, const U2& u2, ..., const UN& uN);

    bool is_open() const;
    void close();

    T& operator*();
    T* operator->();
};
```

解说

stream 是标准库流的子类，因此具有标准 IO 流的所有能力，例如 get()、read() 等操作。它实际上有四个模板参数，但只有第一个流要连接的设备类型 T 是必须的，其他的都可以使用缺省值。stream 并不像标准库的流那样用名字来区分输入流和输出流，它的方向完全取决于它关联的设备 T——使用源设备就是输入流，使用接收设备就是输出流。

stream 可以在构造的时候直接打开设备，也可以稍后调用 open()函数打开设备，它们的函数参数很有些特点：单参的形式接受设备的实例，多参版本则接受构造设备所需的 N 个参数，由 stream 在内部创建设备的实例。正因为如此，我们之前的代码中才可以直接传递数组首末地址来直接创建流。

如果流已经关联到实际的设备（构造时打开或者调用 open()），那么成员函数 is_open()会返回 true。这时可以使用重载的 operator*和 operator->来获得流内部的设备引用，就像是一个智能指针。

基本流的用法之前已经演示了很多，故这里不再举例。

8.6.2 过滤流

过滤流可以说是基本流的强化版，它不仅能够连接设备，更可以连接过滤器和链，因此，它的用途更加广泛，功能更加强大。

在 iostreams 库中过滤流有 filtering_stream 和 filtering_streambuf 两个模板类，两者的功能和行为很相似，为简单起见，我们只讨论 filtering_stream，它位于头文件<boost/iostreams/filtering_stream.hpp>。

类摘要

与 stream 一样，filtering_stream 使用了模板元编程技术，根据模板参数信息推导出它的父类，可能是 std::basic_istream、std::basic_ostream 或者 std::basic_iostream，我们之前使用的 filtering_ostream 和 filtering_istream 实际上是它的模式特化类，模式分别为 output 和 input。

filtering_stream 简化的摘要如下：

```
template< typename Mode>
class filtering_stream  {
public:
```

```
    typedef Ch                    char_type;    //字符类型
    typedef Mode                   mode;        //过滤器的模式

    filtering_stream();
    template<typename T>
    filtering_stream( const T& t);
    template<typename StreamOrStreambuf>
    filtering_stream( StreamOrStreambuf& t);

    const std::type_info& component_type(int n) const;

    template<typename T>
    T* component(int n) const;

    template<typename T>
    void push( const T& t);
    template<typename StreamOrStreambuf>
    void push( StreamOrStreambuf& t);

    void pop();
    bool empty() const;
    size_type size() const;
    void reset();
    bool is_complete() const;
private:
    Chain chain_;                                //设备链
};
```

解说

虽然过滤流是基本流的强化版，但比较代码可以发现两者之间的差异还是较大的，filtering_stream 的接口与 chain 更接近一些。

filtering_stream 的主要模板参数是 Mode，指定了过滤流的模式。构造函数与 stream 差别很大，它不仅能够接受设备，还能够接受设备链和流，这一点请读者务必注意。

filtering_stream 内含了 chain 的实现，因此它具有与 chain 相似的接口，可以向链中添加、删除或者访问设备，以及检查链是否完整。

用法

与 stream 不同，filtering_stream 不能直接连接设备来决定它的模式，而是需要使用模板参数指明，但我们通常会直接使用定义好的 filtering_istream 和 filtering_ostream。

过滤流的构造函数可以传入多个设备对象，中间用管道符（“|”）连接，形成一个设备链。如果终点是一个接收设备那么它就相当于一个增强了的输出流，可以被写入，写入数据时执行过滤操作，反之则是一个增强的输入流。但需要注意，管道操作只能针对设备，它不能连接流，如果要把流对象加入过滤流则必须使用成员函数 push()。

示范过滤流用法的代码如下：

```
#include <boost/iostreams/stream.hpp>
#include <boost/iostreams/device/array.hpp>
#include <boost/iostreams/copy.hpp>
#include <boost/iostreams/filtering_stream.hpp>
#include <boost/iostreams/filter/counter.hpp>

int main()
{
  char ar[10] = "abcd";
  stream<array_source> sa(ar);    //定义一个字符数组输入流

  counter ct;                     //一个计数过滤器
  filtering_istream in;           //定义一个空的输入过滤流
  assert(in.empty());
  in.push(ref(ct));               //加入计数过滤器，使用 ref 包装，链不完整
  assert(!in.is_complete());
  in.push(sa);                    //加入输入流，链完整
  assert(in.is_complete());

  io::copy(in,                    //io::copy 算法从输入过滤流拷贝
      filtering_ostream(cout)     //输出过滤流直接连接标准输出，没有添加过滤器
      );

  cout << ct.characters();
}
```

这段代码示范了 filtering_stream 的多个成员函数的使用，需要注意的是输入输出

流的末端都是流而不是设备。

8.7 流处理函数

设备、过滤器和流是 iostreams 库的三大要件，但仅有这三个概念还是不够的，流虽然把设备和过滤器连接在一起，但它们仍然是静止的，要想让数据真正地流动起来还需要一个外部的驱动——流处理函数。

iostreams 库提供了数个流处理函数，其中最重要的是 **io::copy()**，它驱动了整个数据流——从源设备或流中读入字符，再写入到接收设备或流中，最后关闭两个设备并返回处理的字符数。

io::copy()位于头文件<boost/iostreams/copy.hpp>，声明形式如下：

```
template<typename Source, typename Sink>
std::streamsize copy( Source& src, Sink& sink);
```

io::copy()有四种重载形式，用来应对 src 和 sink 是设备或流的情形。我们之前已经多次使用了 io::copy()，但都是操作流对象，实际上它也可以直接操作设备，有的时候会使代码更加简单，例如：

```
char ar[] = "abcd123";                          //字符数组
string result;                                  //标准字符串
io::copy(stream<array_source>(ar), cout);//从源设备到输出流
io::copy(stream<array_source>(ar),              //从源设备到接收设备
    io::back_inserter(result));
```

除了 io::copy()，iostreams 库中还有其他流处理函数，它们大都位于头文件<boost/iostreams/operations.hpp>，常用函数如下：

- get() ：类似标准流的成员函数，以一致的方式从源设备或流中获取一个字符；
- read() ：类似标准流的成员函数，以一致的方式从源设备或流中获取多个字符；
- put() ：类似标准流的成员函数，以一致的方式向接收设备或流中写入一个字符；
- write()：类似标准流的成员函数，以一致的方式向接收设备或流中写入多个字符；
- seek() ：类似标准流的成员函数，以一致的方式随机访问设备或流；
- flush()：类似标准流的成员函数，以一致的方式刷新设备或整个流，清空缓冲区；

■ close()：类似标准流的成员函数，以一致的方式关闭设备或流。

这些函数的声明如下：

```
template<typename T>
typename int_type_of<T>::type
get(T& t);

template<typename T>
inline std::streamsize
read(T& t, typename char_type_of<T>::type* s, std::streamsize n);

template<typename T>
bool
put(T& t, typename char_type_of<T>::type c);

template<typename T>
inline std::streamsize
write(T& t, const typename char_type_of<T>::type* s, std::streamsize n);

template<typename T>
inline std::streampos
seek( T& t, stream_offset off, seekdir way);

template<typename T>
bool flush(T& t);

template<typename T>
void close(T& t);
```

应用流处理时我们通常很少使用这些函数，它们主要在编写自定义设备或过滤器时发挥作用，使得我们能够以一致的方式方便地编写泛型代码来操作流。

8.8 定制设备

通过前几节的讨论，我们已经基本熟悉了 iostreams 库的各个组件和用法，能够操作预定义的设备、过滤器和流执行流处理，但 iostreams 库不仅仅是一个工具库，它更是一个框架，允许——更进一步地说是鼓励我们在这个框架之内编写自己的代码，去扩充、增强流处理的能力。

要编写可用于 iostreams 流处理框架的类，我们必须对 iostreams 库的设备、过滤器、流等概念有一定程度的了解。好在我们不必白手起家，iostreams 库预定义的若干设备和过滤器都是良好的范例，在熟悉它们功能的基础上阅读其实现源代码是一个很好的提高自身水平的方法。

对 iostreams 的扩展可从编写设备和编写过滤器这两个方面入手，本节研究编写设备，下一节研究编写过滤器。

8.8.1 定制源设备

编写设备首先要满足设备的概念（参见 8.4.1 小节），具有 char_type 和 category 内部类型定义，对于源设备来说其模式则必须为 input。

实现原理

device 类在头文件<boost/iostreams/concepts.hpp>中定义了若干特化表示一些设备子概念，其中就包括为了方便用户扩展而定义的 source，在此再声明如下：

```
typedef device<input>    source;                 //char 源设备
```

只要我们自定义的设备 public 继承 source，那么设备就会自动满足源设备的概念。当然，不一定非要使用 source 作为基类(例如 iostreams 库自己的数组设备和文件设备)，但对于用户来说通常这样会更加方便，易于上手。

源设备被用于输入，因此它需要实现一个如下形式的 read()成员函数：

```
std::streamsize read(char_type* s, std::streamsize n);
```

read()函数读取最多 n 个字符到缓冲区 s 中，然后返回读取的字符数表示读取成功，返回 EOF（-1）表示已经读取完毕。

示例 1：随机数源设备

这里我们使用 boost.random 库的 rand48 随机数发生器编写一个随机数源设备 rand_source，它可以产生从'0'到'z'的随机字符，实现代码如下：

```
#include <boost/random.hpp>                //boost.random 库

//使用变量发生器组合 rand48 和 uniform_smallint
typedef variate_generator<rand48, uniform_smallint<>> rand_t;

class rand_source: public io::source      //定义随机数源设备
```

```
{
private:
  static rand_t rand;                    //随机数源，静态成员变量
  int count;                             //需要产生的随机数数量
public:
  rand_source(int c):                    //构造传入数量
      count(c){   }

  //核心函数，实现源设备的流读取功能
  std::streamsize read(char_type* s, std::streamsize n)
  {
      using namespace std;
      // 读取最多 n 个字符到缓冲 s 中，返回读取的字符数
      streamsize readCount = (min)(n, count);
      if (readCount)                     //应读取的数量
      {
          for (streamsize i = 0;i < readCount; ++i)
          {
              *s++ = rand();             //把随机数拷贝到缓冲区
          }
          count -= readCount;            //当前剩余的读取数量
          return readCount;              //返回读取的字符数
      }
      else
      {
          // 返回 EOF 表示已经读取完毕
          return EOF;
      }
  }   //read()成员函数结束
};

//静态成员变量的定义
rand_t rand_source::rand(rand48(time(0)), uniform_smallint<>('0', 'z'));
```

有了这个源设备，我们就可以把它作为 stream 的模板参数，创建一个流用于流处理：

```
int main()
{
  string out;                          //输出用标准字符串
```

```
  io::copy(stream<rand_source> (20), //创建流，直接传源设备的构造参数
      io::back_inserter(out));       //输出到字符串
  assert(out.size() == 20);          //生成了 20 个随机字符
}
```

示例 2：改进的随机数源设备

第二个例子我们对 rand_source 做出一点小小的改动，把流处理的字符类型由 char 变为 unsigned char，因为 source 是 char 设备，因此我们不能从它继承，需要从 device 直接特化，其他的逻辑则不变：

```
typedef variate_generator<rand48, uniform_smallint<>> rand_t;
class rand_source: public io::device<input, unsigned char>
{...};     //实现代码同前
rand_t rand_source::rand(rand48(time(0)), uniform_smallint<>(0, 255));
```

因为流处理设备链上的设备必须使用一致的字符类型，所以我们再定义一个配套的标准容器 block，它是 basic_string 的特化：

```
typedef std::basic_string<unsigned char> block;
```

新的 rand_source 的用法与原来相同，但它处理的是 unsigned char 字符类型：

```
block out;                     //注意，不能使用 std::string，因为字符类型不同
io::copy(rand_source(100),     //直接从源设备拷贝
    io::back_inserter(out));
assert(out.size() == 100);
```

8.8.2 定制接收设备

如果理解了源设备的编写原理，那么我们也可以很快地学会编写接收设备。自定义接收设备同样要满足 device 的定义，具有 char_type 和 category 内部类型定义，只不过对于接收设备来说其模式则必须为 output，它的便捷特化形式是 sink：

```
typedef device<output>  sink;   //char 接收设备
```

对于接收设备，我们需要编写如下形式的 write() 成员函数：

```
std::streamsize write(const char_type* s, std::streamsize n)
```

write() 从缓冲区 s 中读取最多 n 个字符，然后写入某个地方（文件、内存等），最后

返回写入的字符数，处理逻辑比源设备的 read()要简单一些。

下面的代码实现了一个非常简单的接收设备 my_sink，它仅仅把字符输出到标准流上：

```
class my_sink: public io::sink
{
public:
  streamsize write(const char_type* s, streamsize n)
  {
      cout << string(s,n) << endl;
      return n;
  }
};
```

my_sink 的测试代码如下：

```
int main()
{
  string str("abcd");
  io::copy(
      boost::make_iterator_range(str),
      my_sink()); //直接向接收设备输出
}
```

8.9　定制过滤器

过滤器是 iostreams 真正展现威力的地方，流处理大多数有用的功能都是通过过滤器实现的，因此，编写自己的过滤器是使用 iostreams 库的必备技能。

iostreams 库中预定义的过滤器都是很好的范例，读者可以参照研究。

8.9.1　过滤器实现原理

编写过滤器必须符合 filter 的概念，它位于头文件<boost/iostreams/concepts.hpp>。

根据模式的不同，过滤器可以分为输入过滤器、输出过滤器、两用过滤器、可定位过滤器等，最常用的是前两者，分别用于输入流和输出流。根据访问字符的方式过滤器又可分为普通（单字符）过滤器和多字符过滤器，普通过滤器一次只处理一个字符，虽然效率低但较容易处理，而多字符过滤器则正好相反，接下来我们主要编写多字符过滤器。

过滤器辅助类

iostreams 库提供了数个编写过滤器的辅助类，让用户可以更容易地编写自己的过滤器，这些过滤器包括：

- aggregate_filter ：两用过滤器，一次性接收所有的数据；
- basic_line_filter ：两用过滤器，每次提取一行数据；
- basic_stdio_filter：两用过滤器，用于适配标准输入输出流；
- symmetric_filter ：两用过滤器，带有内部缓冲区。

四个过滤器中前三个用起来比较简单，我们只需要遵循模板方法模式实现纯虚函数 do_filter()，在里面编写适当的处理逻辑就可以了，而 symmetric_filter 应用较复杂，本书暂不做介绍。

管道符

让自定义过滤器支持 iostreams 的管道符操作通常是个好主意，它可以让过滤器更好地搭配其他设备和流工作。iostreams 库特别提供了一个宏 BOOST_IOSTREAMS_PIPABLE，它可以只用一行代码就给过滤器增加管道能力：

```
#define BOOST_IOSTREAMS_PIPABLE(filter, arity)
```

BOOST_IOSTREAMS_PIPABLE 通常用在过滤器定义之后，它的第一个参数 filter 是过滤器的名字，第二个参数 arity 是过滤器模板参数的数量，如果过滤器不是模板类，那么值应该是 0。

BOOST_IOSTREAMS_PIPABLE 使用了预处理元编程技术，宏展开后会形成一个针对 filter 重载的 operator|函数，因此宏后面无须添加分号（当然，加上分号也不算错误）。

8.9.2 aggregate_filter

aggregate_filter 可以一次性读取全部字符序列，它位于头文件<boost/iostreams/filter/aggregate.hpp>，类摘要如下：

```
template< typename Ch>
class aggregate_filter  {
public:
    typedef std::vector<Ch>      vector_type;
```

```
private:
    virtual void do_filter(const vector_type& src, vector_type& dest) = 0;
    virtual void do_close() { }
};
```

aggregate_filter 是一个两用过滤器，这意味着只要使用它就既可以实现输入过滤器也可以实现输出过滤器。它使用模板方法模式完成了过滤器所需的大部分读写工作，只需要实现过滤的核心功能 do_filter()纯虚函数。它有两个参数，src 是过滤器接收到的所有字符，我们可以对这些字符进行任意的处理，然后把处理结果添加到 dest，这两个参数都存储在 std::vector<Ch>中。

另一个虚函数 do_close()在过滤器被关闭时调用，通常可以用缺省的空实现，但有的时候也可以重载它做一些特别的关闭操作。

8.5.4 小节介绍的 regex_filter 就是利用 aggregate_filter 实现的。

示例：sha1 过滤器

作为示范，我们使用 aggregate_filter 来编写一个计算字符序列 sha1 摘要的输出过滤器 sha1_filter，它使用 boost.uuid 库的 sha1 类计算摘要，代码如下：

```
#include <boost/iostreams/filter/aggregate.hpp>  //包含过滤器头文件
#include <boost/uuid/sha1.hpp>                    //sha1 摘要功能

template<typename Ch = char>                      //模板参数，默认是窄字符
class sha1_filter:
  public io::aggregate_filter<Ch>                 //使用 aggregate_filter
{
private:
  //实现虚函数，过滤器的核心功能
  virtual void do_filter(const vector_type& src, vector_type& dest)
  {
      using namespace boost::uuids::detail;       //sha1 的名字空间

      sha1 sha;                                   //sha1 对象
      sha.process_bytes(&src[0], src.size());     //处理所有输入字符序列

      unsigned int digest[5];                     //获得摘要值
      sha.get_digest(digest);
```

```
        Ch* p = reinterpret_cast<Ch*>(digest);          //整型转换为 char 类型
        std::copy(p, p + 20, std::back_inserter(dest));  //拷贝输出
    }
};
BOOST_IOSTREAMS_PIPABLE(sha1_filter, 1)                 //增加管道功能实现
```

下面的代码示范了这个新过滤器的使用：

```
int main()
{
    char str[] = "123";                                 //字符数组
    string result;                                      //标准字符串

    filtering_ostream out(                              //输出流
        sha1_filter<>() |                               //sha1 过滤器
        counter() |                                     //统计流过的字符数
        io::back_inserter(result));                     //用 string 接收摘要

    io::copy(stream<array_source>(str, str + 3), out);  //流处理
    assert(result.size() == 20);                        //sha1 摘要长度为 20 字节
    assert(out.component<counter>(1)->characters() == 20);
}
```

8.9.3 basic_line_filter

basic_line_filter 是一个类似 aggregate_filter 的过滤器辅助类，它位于头文件<boost/iostreams/filter/line.hpp>，类摘要如下：

```
template<typename Ch>
class basic_line_filter {
private:
    virtual string_type do_filter(const string_type& line) = 0;
};
```

basic_line_filter 是个两用过滤器，用法也与 aggregate_filter 很像，但它一次过滤一行字符，纯虚函数 do_filter() 处理这行字符，再返回处理后的新行。

为了方便使用，basic_line_filter 提供了 line_filter 和 wline_filter 两个特化：

```
typedef basic_line_filter<char>     line_filter;
```

```
typedef basic_line_filter<wchar_t>  wline_filter;
```

8.5.5 小节的 grep_filter 就是利用 basic_line_filter 实现的。

示例：简单的行过滤器

下面的代码实现了一个示范性质的行过滤器 my_line_filter，它的功能非常简单，为每一行首尾两端增加一对尖括号：

```
#include <boost/iostreams/filter/line.hpp>
template<typename Ch = char>
class my_line_filter:
  public io::basic_line_filter<Ch>           //使用 basic_line_filter
{
private:
  virtual string_type do_filter(const string_type& line)
  {
      return "<" + line + ">";               //简单处理一下行然后返回
  }
};
BOOST_IOSTREAMS_PIPABLE(my_line_filter, 1);  //增加管道功能实现
```

my_line_filter 可以这样使用：

```
int main()
{
  string str("123\nxyz\nmonado");

  io::copy(
      filtering_istream(                     //使用输入过滤流
          my_line_filter<>() |               //行过滤器
          boost::make_iterator_range(str)),  //从标准字符串输入
      cout                                   //流出到标准输出流
      );
}
```

这段代码中需要注意的是我们把 my_line_filter 加入到了输入流中，因为 my_line_filter 是个两用流，所以它允许这样使用。

8.9.4　手工打造过滤器

之前我们都使用的是 iostreams 库提供的过滤器辅助类来编写过滤器，但有的时候这

些辅助类并不能满足我们的要求，我们还需要完全手工编写过滤器类。

手工编写过滤器其实并不复杂，只是把原来辅助类替我们实现的 char_type、category、read()、write()、close()等过滤器必备要素自己编写出来就可以了。因为这些都由我们自己控制，所以可以获得更多的灵活性。

对于多字符过滤器来说，我们需要实现如下形式的读写函数：

```
template<typename Source>
std::streamsize read(Source& src, char_type* s, std::streamsize n)
{
    //从 src 读取字符处理，处理后写入最多 n 个到缓冲区 s
    //返回读取的字符数或者 EOF
 }

template<typename Sink>
std::streamsize write(Sink& dest, const char_type* s, std::streamsize n)
{
    //从缓冲区 s 读取最多 n 个字符处理，将处理后的字符写到 dest
    //返回已处理的字符数
}
```

示例：另一个 sha1 过滤器

我们把 8.9.2 小节实现的 sha1 过滤器重新实现为一个输出过滤器 sha1_filter_ex 如下：

```
#include <boost/uuid/sha1.hpp>          //sha1 摘要功能
class sha1_filter_ex  //不是模板类，也不使用任何 iostreams 库的概念类
{
public:
   typedef char char_type;              //字符类型定义
   struct category:                     //过滤器分类定义
       output,                          //输出模式
       filter_tag,                      //过滤器设备
       multichar_tag,                   //处理多字符
       closable_tag                     //可关闭
       { };
private:
   boost::uuids::detail::sha1 sha;      //sha1 对象
```

```
public:
  virtual ~sha1_filter_ex(){}         //虚析构函数

  template<typename Sink>             //输出模式需要实现 write()函数
  std::streamsize write(Sink&, const char_type* s, std::streamsize n)
  {
      sha.process_bytes(s, n);        //处理所有输入字符序列
      return n;                       //返回处理的字符数，但暂不输出摘要结果
  }

  template<typename Sink>
  void close(Sink& snk)               //过滤器关闭时给出摘要结果
  {
      unsigned int digest[5];         //获得摘要值
      sha.get_digest(digest);

      char_type* p = reinterpret_cast<char_type*>(digest);
      io::write(snk, p, sizeof(digest)/sizeof(char_type));   //写入接收设备
  }
};
BOOST_IOSTREAMS_PIPABLE(sha1_filter_ex, 0);                  //支持管道符操作
```

请读者注意 sha1_filter_ex 与 sha1_filter 的区别，它没有从任何过滤器概念继承，因此需要手工定义它的 char_type 和 category，为了方便我们没有使用模板参数，而是直接把 char 定义为它使用的字符类型，过滤器的分类则是可处理多字符的、可关闭的输出过滤器。

sha1_filter_ex 的模式决定了我们要实现的成员函数。write()中执行摘要操作，但与 sha1_filter 不同的是它过滤后并不向接收设备输出，而是在 close()关闭过滤器时才输出，因此我们可以向 sha1_filter_ex 多次传入待摘要的数据，最后关闭过滤器获得摘要结果。

示范 sha1_filter_ex 用法的代码如下：

```
int main()
{
  char str[] = "123456";                   //字符数组
  string result;                           //标准字符串

  filtering_ostream out(                   //用于输出流
```

```
        sha1_filter_ex() |                          //sha1 过滤器
        io::back_inserter(result));                 //用 string 接收摘要

    io::write(out, str, 3);                         //使用 write()函数向流写入数据
    assert(result.size() == 0);                     //此时没有输出结果

    io::write(out, str + 3, 3);                     //继续使用 write()函数向流写入数据
    assert(result.size() == 0);                     //此时仍然没有输出结果

    io::close(out);                                 //关闭流
    assert(result.size() == 20);                    //得到摘要结果
}
```

为了示范 sha1_filter_ex 多段摘要的能力，代码中我们没有使用 io::copy()函数，这是因为 io::copy()不仅拷贝了流数据还会自动关闭流，故 copy 后就无法再使用流了。而 write()函数则不会关闭流，所以我们可以向流多次写入数据，最后手动关闭流获得结果。

sha1_filter_ex 也可以使用另一个概念类 multichar_output_filter 来简化代码，它仅定义了 char_type 和 category，代码如下所示：

```
class sha1_filter_ex: public io::multichar_output_filter
{
        //无须再定义 char_type 和 category
    ... //其他同前
}
```

实现不关闭设备的 copy 算法

对于 sha1_filter_ex 这样的可多段输入的过滤器来说不能使用 io::copy()算法，而 write()用起来显然没有 io::copy()方便，可惜 iostreams 库并没有提供不自动关闭设备的 copy 版本。下面我们就仿照 io::copy()编写一个 copy_no_close()函数，限于篇幅仅实现了一个对流操作的版本，其他操作设备的版本读者可自行实现。

io::copy()算法的核心是函数对象 copy_operation，并使用模板元编程技术来处理 source 和 sink 的各种可能情况，我们的版本则简化了很多：

```
//使用辅助函数来自动推导模板参数
template<typename Source, typename Sink>
std::streamsize
copy_no_close_impl(Source src, Sink snk,
```

```
            std::streamsize buffer_size )
{
  return io::detail::copy_operation<Source, Sink>(
      src , snk, buffer_size )
      (); //调用 copy 操作，不关闭设备
}

//调用辅助函数完成拷贝操作
template<typename Source, typename Sink>
std::streamsize
copy_no_close( const Source& src, Sink& snk,
            std::streamsize buffer_size =
                   io::default_device_buffer_size
            BOOST_IOSTREAMS_ENABLE_IF_STREAM(Source)
            BOOST_IOSTREAMS_ENABLE_IF_STREAM(Sink) )
{
  return copy_no_close_impl(
      io::detail::wrap(src),
      io::detail::wrap(snk),
      buffer_size );
}
```

copy_no_close 的用法与 io::copy()基本相同，记得最后要使用 io::close()：

```
copy_no_close(stream<array_source>(str, str + 3), out);
io::close(out);
```

8.10 组合设备

现在我们已经初步具备了编写自定义设备和过滤器的能力，但是很多情况下复杂的设备和过滤器的实现难度很大，因此 iostreams 库提供了一些模板类和函数，可以把简单易实现的设备或过滤器组合起来，形成可用的新设备。

8.10.1 combine

模板类 combination 可以组合一对源/接收设备或者输入/输出过滤器，形成一个新的设备或者过滤器。新的设备是一个可在两个独立字符序列上工作的双向设备，使用前者输入同时使用后者输出。iostreams 库同时提供工厂函数 combine()来简化类的构造，它们位于头文件<boost/iostreams/combine.hpp>。

模板类 combination 和函数 combine() 的声明如下：

```
template<typename In, typename Out>
class combination;

template<typename In, typename Out>
combination<In, Out> combine(const In& in, const Out& out);
```

使用 combine 工具我们可以很容易地创建双向设备，只需要分别实现两个单向的输入输出设备，再使用 combine() 把它们组合起来就可以了。

下面的代码使用 combine() 组合了数组设备和标准容器设备：

```
int main()
{
  char src[] = "12345678";                  //输入字符序列
  string result;                            //输出字符序列

  typedef io::combination<array_source,     //组合设备类型定义
      back_insert_device<string> > bi_dev;

  assert(is_device<bi_dev>::value);         //元函数检查是否是设备
  assert((is_same<                          //元函数检查是否是双向模式
      mode_of<bi_dev>::type,
      bidirectional
      >::value));

  bi_dev dev = io::combine(                 //使用函数生成组合设备
      array_source(src),
      io::back_inserter(result));

  io::write(dev, src, 2);                   //向输出序列写入数据
  assert(result.size() == 2);
}
```

8.10.2 compose

compose 是另一种组合设备或过滤器的工具，它把两个设备“串联”起来，数据顺序（输出模式）或逆序（输入模式）流过新的设备，颇有些类似管道符的作用，它位于头文件

<boost/iostreams/compose.hpp>。

compose 提供一个模板类 composite，它的第一个模板参数是过滤器，第二个模板参数是过滤器、设备或者流，类似 std::pair 可以用 first() 和 second() 获得它组合的对象，工厂函数 compose() 用来自动推导参数生成新设备对象，它们的声明如下：

```
template<typename Filter, typename FilterOrDevice>
class composite {
public:
    typedef typename char_type_of<Filter>::type  char_type;
    composite(const Filter& first, [const] FilterOrDevice& second);

    Filter& first();
    Device& second();
};

template<typename Filter, typename FilterOrDevice>
composite<Filter, FilterOrDevice>
compose(const Filter& first, [const] FilterOrDevice& second);
```

下面的代码简单示范了 compose 的用法，它把正则表达式过滤器和标准输出流组合在了一起，效果与 8.5.4 小节完全相同：

```
int main()
{
  string str("abcdef aochijk");                        //输入字符串

  io::copy(                                            //io::copy 算法流处理
      boost::make_iterator_range(str),
      compose(regex_filter(regex("a.c"), "test"),      //正则表达式过滤器
          cout)                                        //标准输出流
      );
}
```

8.10.3　tee

tee 工具提供了分离流输出的功能，它可以建立流的分支，把上游的数据同时发给另外一个接收设备，配合 compose 就可以实现对一个序列同时执行两种或两种以上不同的流处理，它位于头文件<boost/iostreams/tee.hpp>。

tee 支持两种用法，一种是把接收设备适配为输出过滤器，可以加到设备链的中间；另一种是把两个接收设备组合成一个接收设备，加到设备链的末尾。因此它也就有两个模板类和两个工厂函数，其声明如下：

```
template<typename Sink>
class tee_filter {
public:
    typedef typename char_type_of<Sink>::type  char_type;
    explicit tee_filter([const] Sink& snk);
};

template<typename Sink1, typename Sink2>
class tee_device {
public:
    typedef typename char_type_of<Sink1>::type  char_type;
    tee_device([const] const& sink1, [const] Sink2& sink2);
};

template<typename Sink>
tee_filter<Sink> tee([const] Sink& snk);
template<typename Sink1, typename Sink2>
tee_device<Sink1, Sink2> tee([const] Sink1& sink1, [const] Sink2& sink2);
```

下面的代码把 8.10.2 小节的例子做了一点变动，把 compose 形成的新接收设备适配成了 tee 输出过滤器，而原来的数据则流经 sha1 过滤器，这样数据将同时输出到标准流和标准容器：

```
int main()
{
  string str("abcdef aochijk");           //输入字符串
  string result;                          //标准容器接收设备

  io::copy(                               //io::copy 算法流处理
      boost::make_iterator_range(str),    //源设备
      filtering_ostream(                  //输出流
          tee(                            //tee 的过滤器用法
              compose(regex_filter(regex("a.c"), "test"),//compose
                  cout))
          | sha1_filter()|                //管道符
```

```
        io::back_inserter(result)                           // 标准容器接收设备
    ));
}
```

tee 的第二种用法如下：

```
io::copy(                                                   //io::copy算法流处理
    boost::make_iterator_range(str),                        //源设备
    tee(                                                    //tee 的设备用法
        compose(regex_filter(regex("a.c"), "test"),         //compose
                cout),
        io::back_inserter(result)
        )
    );
```

8.11 其他议题

本小节简要讨论关于 iostreams 库的一些其他议题。

流处理的元函数

iostreams 库使用了较多的模板元编程技术，这里对之前涉及的若干元函数做一个归纳：

- char_type_of<T>：以::type 返回设备的字符类型；
- mode_of<T>　　　：以::type 返回设备的模式；
- category_of<T> ：以::type 返回设备的分类信息；
- is_device<T>　 ：以::value 返回类型是否是一个设备；
- is_filter<T>　 ：以::value 返回类型是否是一个过滤器；

这些元函数可以用在我们的泛型代码中，在编译期保证程序的正确性。

对象的生存周期

如果读者曾经使用过 Crypto++或者 Botan 这两个著名的密码学库，那么可能会知道它们也使用了流处理的思想，也具有 iostreams 库中类似的 source、sink、filter、pipe、chain 等概念，不过它们与 iostreams 库对对象的生存周期管理方式存在着很大

的区别。在 Crypto++或者 Botan 的设备链中可以直接传递 new 生成的设备对象指针，由链来管理设备对象的生存周期，用起来很方便，而 iostreams 库则要求设备必须是可拷贝构造的，否则就要使用 boost.ref 来包装引用，用起来有那么一点不便。

实际上 iostreams 库曾经考虑过动态创建对象并传递所有权的方式，并且有过基于这一方式的实现，但最后考虑到异常安全的问题而最终废弃了这种方式，现在的拷贝构造方式对设备虽然有更多的要求但也更加安全。

与迭代器的比较

把 iostreams 库与第 3 章的迭代器放在一起比较会很有意思，下面就对它们做一个简单的对比，可以让我们更好地理解两者。

首先看相似之处，流处理与迭代器对数据序列的处理方式非常相似，都使用一个 copy 算法，遍历一个源序列，然后把处理过的数据写入另一个序列；流的模式与迭代器的分类很接近，也可以分为可读的输入模式和可写的输出模式；迭代器可以嵌套组合提供复杂的功能，而流则可以用设备链的形式过滤处理数据。

流处理与迭代器之间的差异也是很明显的，简单列举如下：

- 迭代器基于迭代器设计模式，不使用虚函数，通过改写迭代器的核心操作来完成对数据的处理；iostreams 则基于已确定的标准流框架，使用流处理思想，并使用设备、过滤器等概念来处理数据；
- 迭代器是泛型的，可以处理任何类型的数据；iostreams 虽然也是泛型的，但它更侧重于处理字符类型；
- 迭代器通常用于处理内存中的数据，而 iostreams 则还可以处理文件、socket 等更广范围的数据；
- 迭代器虽然也可以组合使用，但它只能在编译期确定，显然没有流的设备链可以任意添加拆卸的方式灵活；
- 虽然都使用 copy 算法，但迭代器需要使用一对迭代器来指定拷贝的区间，而流因为可以自动结束所以不需要使用区间的形式，代码更简单。

8.12　总结

本章我们研究了 boost.iostreams 库，它扩展了标准库的流处理能力，提供了一个

更加强大易用的流处理框架。

iostreams 库基于标准库的流实现，而标准流使用了多重继承和虚函数，因此它存在少量的虚函数调用开销，这是不可避免的，但 iostreams 库通过大量的泛型设计和模板元编程技术极大地降低了虚函数的代价，在一般的应用中是绝对可以接受的。

iostreams 库已经被广大的 Boost 用户证明结构清晰并且很容易学习，但因为包含的内容太过庞杂丰富，一些组件的用法作者也没有完全掌握，故书中展现给读者的也只能是其中的一小部分，它的更多的功能还需要读者在实践开发中进一步探索。

本章大致可分为以下三部分：

- 8.1 节至 8.2 节 ：阐述流处理的工作原理和 iostreams 库的组织结构；
- 8.3 节至 8.7 节 ：详细介绍 iostreams 库中的基本概念和组件的使用方法；
- 8.8 节至 8.10 节：介绍基于 iostreams 库的框架自定义设备和过滤器来扩展流处理的功能。

第一部分我们讨论了标准库的流处理框架和 iostreams 库对它的扩展，随后用两个示例程序初步示范了 iostreams 库中的设备、过滤器、流等的用法。

第二部分我们讨论了流处理的核心内容，介绍了设备、过滤器、流等重要概念，并研究了若干 iostreams 库预定义的设备和过滤器。使用 iostreams 库提供的这些设施我们可以很容易地编写流处理代码，以流的方式执行很多有用的功能。

第三部分我们讨论了定制流处理，可以基于 iostreams 库的框架编写自己的设备和过滤器，能够随心所欲地操作各种数据流，几乎一切数据都能以流的方式处理。iostreams 库不仅提供了实现设备和过滤器的基本类，还提供了一些用来把简单的设备组合成复杂设备的辅助类，利用好它们会使代码更加简洁优雅。

第 9 章

序列化

序列化（serialize）对于编程语言来说是一个很有用的功能，它可以把对象转化成一个字节流存储或者传输，在需要时再恢复成与原状态一致的等价对象。很多编程语言都内建实现了序列化功能，但标准 C++对此没有定义。

boost.serialization 以库的形式为 C++提供了这个重要的功能，它非常强大，可以序列化 C++中的各种类型，同时又非常简单易用，对于每一个 C++程序员都是很有价值的工具。

序列化与流处理联系十分紧密，因此读者在阅读本章时最好已经对第 8 章有所了解。

9.1 编译与使用

boost.serialization 库必须编译后才能使用，本节将介绍库的编译和使用方法。

9.1.1 编译

boost.serialization 库可以使用 bjam 直接编译成静态库或动态库，也可以把所有源代码嵌入一个 cpp 文件中编译。bjam 的用法可见 Boost 自带文档或者推荐书目[1]，这里不做详细解说，只介绍嵌入编译的方式。

serialization 库的所有 cpp 文件位于<libs/serialization/src/>下，由于源代码多，故编译源文件也较大：

```
/// sprebuild.cpp
#include <boost/config/warning_disable.hpp>  //消除 VC 警告
```

```
#if defined(_MSC_VER) && (_MSC_VER >= 1400)
#  pragma warning(disable:4267)              //消除 VC 警告
#endif

#define BOOST_ARCHIVE_NO_LIB                    //禁用自动连接，使用源代码
#define BOOST_SERIALIZATION_NO_LIB              //禁用自动连接，使用源代码

// 需修改<boost/archive/impl/archive_serializer_map.ipp>
// 加入 include guard
// 例如：#ifndef BOOST_ARCHIVE_SERMAP_IPP
//       #define BOOST_ARCHIVE_SERMAP_IPP
//       #endif

#include <libs/serialization/src/archive_exception.cpp>
#include <libs/serialization/src/basic_archive.cpp>
#include <libs/serialization/src/basic_iarchive.cpp>
#include <libs/serialization/src/basic_iserializer.cpp>
#include <libs/serialization/src/basic_oarchive.cpp>
#include <libs/serialization/src/basic_oserializer.cpp>
#include <libs/serialization/src/basic_pointer_iserializer.cpp>
#include <libs/serialization/src/basic_pointer_oserializer.cpp>
#include <libs/serialization/src/basic_serializer_map.cpp>
#include <libs/serialization/src/basic_xml_archive.cpp>
#include <libs/serialization/src/extended_type_info.cpp>
#include <libs/serialization/src/extended_type_info_no_rtti.cpp>
#include <libs/serialization/src/extended_type_info_typeid.cpp>
#include <libs/serialization/src/polymorphic_iarchive.cpp>
#include <libs/serialization/src/polymorphic_oarchive.cpp>
#include <libs/serialization/src/shared_ptr_helper.cpp>
#include <libs/serialization/src/stl_port.cpp>
#include <libs/serialization/src/void_cast.cpp>
#include <libs/serialization/src/xml_archive_exception.cpp>

#ifdef _UNICODE        //Unicode 支持
#include <libs/serialization/src/basic_text_wiprimitive.cpp>
#include <libs/serialization/src/basic_text_woprimitive.cpp>
#include <libs/serialization/src/binary_wiarchive.cpp>
#include <libs/serialization/src/binary_woarchive.cpp>
#include <libs/serialization/src/codecvt_null.cpp>
```

```
#include <libs/serialization/src/text_wiarchive.cpp>
#include <libs/serialization/src/text_woarchive.cpp>
#include <libs/serialization/src/utf8_codecvt_facet.cpp>
#include <libs/serialization/src/xml_wgrammar.cpp>
#include <libs/serialization/src/xml_wiarchive.cpp>
#include <libs/serialization/src/xml_woarchive.cpp>
#else
#include <libs/serialization/src/basic_text_iprimitive.cpp>
#include <libs/serialization/src/basic_text_oprimitive.cpp>
#include <libs/serialization/src/binary_iarchive.cpp>
#include <libs/serialization/src/binary_oarchive.cpp>
#include <libs/serialization/src/text_iarchive.cpp>
#include <libs/serialization/src/text_oarchive.cpp>
#include <libs/serialization/src/xml_grammar.cpp>
#include <libs/serialization/src/xml_iarchive.cpp>
#include <libs/serialization/src/xml_oarchive.cpp>
#endif   //_UNICODE
```

对于这个编译源文件（sprebuild.cpp）有以下几点需要说明：

- 使用 VC8 编译会有一些小警告，如类型转换可能会丢失数据和著名的 4996，但不会影响库的使用。如果想避免这些警告，可以使用 config 库提供的头文件 <boost/config/warning_disable.hpp>来消除警告（见 sprebuild.cpp 开头的#include）；
- 因为我们把所有源文件合并在了一起，所以必须修改 Boost 源码<boost/archive/impl/archive_serializer_map.ipp>，为它增加“include guard”来防止多次包含；[①]
- 序列化功能对窄字符和宽字符使用不同的代码，所以要使用条件编译区分是否使用 Unicode。

由于 serialization 库包含的功能比较多，因此预编译文件也可以有选择地删减。比如，如果我们只使用序列化为文本的功能，那么就可以把 binary 和 xml 相关的源代码编译注释掉，这样可以减少编译后的代码体积，也加快了编译的速度。

① 这或许是库作者的一个小疏漏：任何可能被包含的源文件——不仅仅是头文件——都应该加入 include guard。

9.1.2 使用

serialization 库的结构与其他 Boost 组件不太一样，头文件被分别放在了两个目录下：<boost/archive/>目录里的头文件处理序列化的存档表现形式，<boost/serialization/>目录里的头文件提供对各种数据类型（如 STL 容器）的序列化能力。

serialization 库主要功能位于名字空间 boost::archive，但因为没有一个统一的头文件供用户使用，所以我们必须根据需要包含特定的存档头文件和序列化头文件。

序列化为纯文本格式需使用头文件如下：

```
#define BOOST_SERIALIZATION_NO_LIB
#include <boost/archive/text_iarchive.hpp>       //文本格式输入存档
#include <boost/archive/text_oarchive.hpp>       //文本格式输出存档
using namespace boost::archive;                  //打开名字空间
```

序列化为 XML 格式需使用头文件如下：

```
#define BOOST_SERIALIZATION_NO_LIB
#include <boost/archive/xml_iarchive.hpp>        //xml 格式输入存档
#include <boost/archive/xml_oarchive.hpp>        //xml 格式输出存档
using namespace boost::archive;                  //打开名字空间
```

序列化为二进制格式（不具有可移植性）需使用头文件如下：

```
#define BOOST_SERIALIZATION_NO_LIB
#include <boost/archive/binary_iarchive.hpp>     //二进制格式输入存档
#include <boost/archive/binary_oarchive.hpp>     //二进制格式输出存档
using namespace boost::archive;                  //打开名字空间
```

9.2 入门示例

与之前的几章一样，对于序列化这样复杂的库仍然使用几个简单的示例程序，用来演示库的基本功能和用法。

9.2.1 示例 1

示例 1 非常短小，它示范了最简单的序列化功能和用法：

```
#define BOOST_SERIALIZATION_NO_LIB              //不使用自动链接技术
#include <boost/archive/text_oarchive.hpp>      //文本格式输出存档
using namespace boost::archive;                 //打开名字空间

int main()
{
  text_oarchive oa(cout);                       //声明一个文本输出存档，连接到 cout 流

  assert(!text_oarchive::is_loading::value);   //不能读出
  assert(text_oarchive::is_saving::value);     //只能写入

  int i = 1;                                    //一个整型变量
  oa << i;                                      //序列化到输出存档
}
```

代码的前三行告诉编译器我们要把 C++对象序列化为纯文本格式，由于我们使用了把源码嵌入工程的方式，所以需要定义宏 BOOST_SERIALIZATION_NO_LIB，避免 VC 的自动链接。

main()函数的第一行代码如下：

```
text_oarchive oa(cout);
```

定义了一个文本输出存档 text_oarchive，它的构造函数要求使用一个标准输出流（std::ostream&），这里我们使用了 cout，因此序列化后的文本将直接输出到屏幕上。

text_oarchive 内部有两个值元函数 is_loading 和 is_saving，它们标明了存档的基本属性，是否可读出或者可写入。由于 text_oarchive 是一个用于输出的存档，所以 is_loading::value == false，而 is_saving::value == true。（参见 9.4 小节）

随后我们声明了一个整数 i，并用流操作符 operator<<把整数序列化到存档中，就像是操作一个过滤流或者文件。代码将会在控制台输出下面一行（颇为神秘的）序列化结果字符串：

```
22 serialization::archive 9 1
```

text_oarchive 不仅重载了 operator<<，而且也重载了 operator&，这是与流不同的地方，所以序列化代码也可以这样写：

```
oa & i;    //效果相同，但没有使用<<那样清晰地表明是序列化输出
```

注意：序列化的操作对象必须是一个左值，我们不能直接序列化常量等右值，否则不能通过编译，如下的写法均是错误的：

```
oa << i+1;     //编译错误
oa << 1;       //编译错误
```

9.2.2　示例 2

示例 1 演示了把 C++对象序列化的初步用法，接下来的第二个例子仍然使用文本格式，演示对 std::string 对象的序列化再反序列化[①]：

```
#define BOOST_SERIALIZATION_NO_LIB
#include <boost/archive/text_iarchive.hpp>    //文本格式输入存档
#include <boost/archive/text_oarchive.hpp>    //文本格式输出存档
using namespace boost::archive;

int main()
{
  ofstream ofs("serial.txt");                 //输出文件流
  string str("serialize test");               //一个标准字符串对象

  {
      text_oarchive oa(ofs);                  //文本输出存档连接到文件流
      oa << str;                              //序列化到输出存档
  }       //存档析构时自动恢复流

  ifstream ifs("serial.txt");                 //输入文件流
  string str2;         //字符串对象用于从存档中反序列化恢复

  {
      text_iarchive ia(ifs);                  //文本输入存档连接到文件流
      ia >> str2;                             //从输入存档反序列化
  }

  assert(str == str2);                        //两个对象等价
}
```

① serialization 库可以直接支持对标准字符串 string 和 wstring 的序列化，不需要特别包含头文件<boost/serialization/string.hpp>。

这段代码中使用了文件流 ofstream 和 ifstream 而不是 cout，这样我们就可以把序列化后的字节流存储起来实现对象的持久化，可以在序列化后的任意时刻反序列化恢复对象。

序列化的代码与示例 1 相同，但我们使用了一对花括号{}来限定了 text_oarchive 对象的生命周期，让它在析构的时候自动关闭对文件流的使用并且恢复流的状态，方便我们接下来使用 ifstream 再次打开文件。

反序列化的代码与序列化的代码类似，只是存档对象改为使用 text_iarchive 连接到文件流 ifstream，并使用流输入操作符 operator>>来恢复对象。当然，operator>>也可以使用 operator&来代替。

如果觉得花括号的使用显得有些麻烦，我们也可以直接使用临时存档对象，让它完成序列化或反序列化工作后就立即销毁，这样代码会更简洁。

```
//构造临时输出存档对象，连接到文件流，序列化输出，然后自动恢复流
text_oarchive(ofs) << str;
//构造临时输入存档对象，连接到文件流，输入反序列化，然后自动恢复流
text_iarchive(ifs) >> str2;
```

文本文件 serial.txt 中存储的是序列化后的 string 对象，内容为：

```
22 serialization::archive 9 14 serialize test
```

9.2.3 示例 3

serialization 库不仅可以使用标准流，也可以使用第 8 章“流处理库”的自定义流，例如：

```
char buf[100];                                  //字符缓冲区
stream<array_sink> sa(buf);                     //使用数组设备的自定义流

text_oarchive oa(sa);                           //文本格式输出存档
int i = 1;
oa << i;                                        //序列化结果存储到字符缓冲区中
```

第三个示例我们改为使用二进制格式存档，并且使用 iostreams 库的压缩过滤器（参见 8.5.6 小节）和过滤流（参见 8.6.2 小节）压缩序列化的结果：

```
#define BOOST_IOSTREAMS_SOURCE
#include <boost/iostreams/filter/zlib.hpp>      //zlib 压缩过滤器
#include <boost/iostreams/device/file.hpp>      //文件设备
```

```
#include <boost/iostreams/filtering_stream.hpp>  //过滤流
using namespace boost::iostreams;

#define BOOST_SERIALIZATION_NO_LIB
#include <boost/archive/binary_iarchive.hpp> //二进制格式输入存档
#include <boost/archive/binary_oarchive.hpp> //二进制格式输出存档
#include <boost/serialization/vector.hpp>    //启用 vector 的序列化
using namespace boost::archive;

int main()
{
  vector<string> vec;                         //一个存储 string 的 vector 容器
  boost::assign::push_back(vec)               //添加若干数据
      ("mario")("luigi")("zelda")("link");

  {
      filtering_ostream ofs(                  //输出过滤流，压缩后存储到文件设备
          zlib_compressor() | file_sink("serial.z"));

      binary_oarchive oa(ofs);                //二进制格式存档，连接到过滤流
      oa << vec;                              //序列化 vector
  }   //自动关闭过滤流

  vector<string> vec2;                        //用来反序列化恢复的 vector

  {
      filtering_istream ifs(                  //输入过滤流，从文件设备读入然后解压缩
          zlib_decompressor() | file_source("serial.z"));

      binary_iarchive ia(ifs);                //二进制格式存档，连接到过滤流
      ia >> vec2;                             //反序列化 vector
  }   //自动关闭过滤流

  assert(vec == vec2);                        //两个对象等价
}
```

这段代码中我们使用了二进制格式的存档，因此需要包含相应的 binary_iarchive.hpp 和 binary_oarchive.hpp，用法上它与文本格式存档没有区别，同样重载了 operator<<和 operator&。

除了这两个二进制存档头文件，我们还多包含了一个用于序列化标准向量容器的头文件 <boost/serialization/vector.hpp>。serialization 库为所有标准容器和部分

Boost 容器都提供了序列化的支持，在序列化这些容器对象时需要自行添加所需的头文件。

存档连接的流我们使用了 filtering_stream，流里添加了一个 zlib 压缩过滤器，它再和源设备或者接收设备连接成一个设备链，在写入或读出时自动地压缩或者解压缩。花括号同样限定了 filtering_stream 的生命周期，这样它可以自动地关闭，不影响后续的文件使用。

我们也可以使用 iostreams 库的流处理函数（参见 8.7 小节）显式地关闭过滤流，例如：

```
filtering_ostream ofs(...);
binary_oarchive (ofs) << vec;
boost::iostreams::close(ofs);
```

9.3 基本概念

serialization 库把存档的格式与类型的序列化完全分离开来，任意的数据类型都可以以采用任意格式的存档保存，带来了极大的灵活性。

本节将介绍 serialization 库的三个基本概念：存档（archive），可序列化（serializable）、序列化（serialize）和反序列化（unserialize）。

9.3.1 存档（archive）

存档（archive）是一个在计算机界常见的词汇，在 serialization 库里表现为一系列的字节（不一定是 ASCII 或者纯二进制），它对应任意的 C++对象，可以持久化保存并在某个时刻恢复成 C++对象。本书中把它译作“存档”，也可以称为“归档”、“档案”。

存档的概念与流很相似，可以依据格式或者输入输出方向进行分类。

依据格式存档可分为以下三类，不同格式的存档不能混用：

- 纯文本格式（text）：序列化后的字节流表现为可阅读的纯文本，可移植性最好；
- XML 格式（xml）：序列化后的字节流表现为可阅读的标准 XML 格式，便于数据交换；
- 二进制格式（binary）：序列化后的字节流表现为不可阅读二进制位，不具有可移植性。

依据输入输出方向存档可分为以下两类，二者互为反操作：

- 输出存档（saving）：可以把 C++对象序列化为某种格式的字节流，元函数 is_saving::value == true，重载了 operator<<和 operator&；
- 输入存档（loading）：可以把某种格式的字节流反序列化为等价的 C++对象，元函数 is_loading::value == true，重载了 operator>>和 operator&。

把这两种分类结合起来，再加上文本表示的 Unicode 支持，就得到了 serialization 库中的所有 10 个存档类型：

- text_oarchive ：窄字符纯文本输出存档；
- text_iarchive ：窄字符纯文本输入存档；
- text_woarchive ：宽字符纯文本输出存档；
- text_wiarchive ：宽字符纯文本输入存档；
- xml_oarchive ：窄字符 XML 输出存档；
- xml_iarchive ：窄字符 XML 输入存档；
- xml_woarchive ：宽字符 XML 输出存档；
- xml_wiarchive ：宽字符 XML 输入存档；
- binary_oarchive：二进制格式输出存档；
- binary_iarchive：二进制格式输入存档。

这些存档类的接口和使用方式都很类似，只是序列化后的字节流表现不同，所以接下来我们以最容易使用同时可移植性最好的 text_oarchive 和 text_iarchive 为主进行介绍。

9.3.2 可序列化

只有可序列化（serializable）的 C++类型才能够被序列化为字节流，保存到存档中或者从存档中恢复。

不是所有 C++类型都是可序列化的，可序列化需要满足下列条件：

- 所有的 C++基本类型都是可序列化的，如 bool、int、double 和 enum；
- 标准字符串 string 和 wstring 是可序列化的，它们被视为基本类型；
- 自定义类如果有特定形式的成员函数或者自由函数 serialize()也是可序列化的；
- 可序列化类型的数组也是可序列化的；
- 可序列化类型的指针和引用也是可序列化的。

上面的五条规则中前两条是可序列化的基础，后三条是对可序列化的扩展和推广。serialization 库以这五条规则构建了几乎 C++所有常用类型的可序列化能力，包括标准库里的复数、valarray 和各种容器，Boost 库里的智能指针和各种新型容器。

如果我们想让自己写的类也支持序列化，只要满足第三条规则就可以了。

9.3.3　序列化和反序列化

序列化（serialize）和反序列化（unserialize）是两个互逆的过程，序列化有时又被称为整编（marshall），反序列化有时又被称为解整编（unmarshall）。

序列化使用输出存档的 operator<<或 operator&把可序列化对象转换为一串字节流，反序列化则使用输入存档的 operator>>或 operator&把字节流恢复成等价的 C++对象。

序列化和反序列化与流的输入输出操作很像，但它们操作的是存档而不是流。

9.4　存档

存档（archive）是 serialization 库的核心概念，本小节将以 text_oarchive 和 text_iarchive 为例介绍关于它的知识。

9.4.1　输出存档

文本格式输出存档 text_oarchive 的类摘要如下：

```
class text_oarchive :
    public text_oarchive_impl<text_oarchive>
{
public:
```

```
    typedef mpl::bool_<false> is_loading;
    typedef mpl::bool_<true> is_saving;

    text_oarchive(std::ostream & os_, unsigned int flags = 0);

    template<class T> Archive & operator<<(T & t);
    template<class T> Archive & operator&(T & t);

    void save_binary(const void *address, std::size_t count);

    library_version_type get_library_version() const;
    unsigned int get_flags() const;

    template<class T>
    some_define* register_type(const T * = NULL);
};
```

输出存档的实现很复杂，但它的接口却很简单：

text_oarchive 使用 mpl 库的 bool 包装器 bool_<>（参见 11.2.3 小节）定义了两个编译期的 bool 标志 is_loading 和 is_saving，标明了存档的输入输出方向，对于 text_oarchive 来说 is_loading::value == false，而 is_saving::value == true。

text_oarchive 的构造函数需要使用一个输出流，很像是对输出流做了一个包装。创建输出存档后我们就可以使用 operator<<或者 operator&向存档写入对象，对象会被自动序列化并流向输出流，成员函数 save_binary()可以对 count 个连续字节的内存执行序列化操作。

text_oarchive 构造函数的第二个参数用于调整存档行为的枚举标志，它可以有如下的取值：

```
enum archive_flags {
    no_header = 1,              //不输出存档的附加头信息，如库版本和签名
    no_codecvt = 2,             //不允许改变流的 codecvt
    no_xml_tag_checking = 4     //不检查 xml 标签是否匹配
};
```

通常我们不需要改变存档的标志位，它对存档的性能几乎不造成什么影响，当前存档使用的标志位可以用 get_flags()获得。

成员函数 get_library_version()返回一个可隐式转换为 uint_least16_t（即 unsigned short）的 library_version_type 对象（声明在<boost/archive/basic_

archive.hpp>)，它标识了当前 serialization 库的实现版本。当前的 Boost1.47 中版本号为 9，而 Boost1.42 中版本号为 7，不同库版本的存档格式可能会有变化，混用不同版本的 serialization 库需要小心。

模板成员函数 register_type<T>()用来向存档"注册"类型 T 的信息，这样在序列化和反序列化时存档可以正确地识别类型，详细的解说见 9.8.4 小节。

9.4.2 输入存档

文本格式输入存档 text_iarchive 的类摘要如下：

```
class text_iarchive :
    public text_iarchive_impl<text_iarchive>,
    public detail::shared_ptr_helper
{
public:
    typedef mpl::bool_<true> is_loading;
    typedef mpl::bool_<false> is_saving;

    text_iarchive(std::istream & is_, unsigned int flags = 0);

    template<class T> Archive & operator>>(T & t);
    template<class T> Archive & operator&(T & t);

    void load_binary(void *address, std::size_t count);

    library_version_type get_library_version() const;
    unsigned int get_flags() const;

    template<class T>
    some_define* register_type(const T * = NULL);
};
```

text_iarchive 的接口与 text_oarchive 很类似，它构造时要求使用一个输入流，操作符重载了 operator<<和 operator&执行反序列化，load_binary()对应输出存档的 save_binary()，恢复连续 count 个字节的数据。

输入存档的标志应该与输出存档的标志一致，否则会因为格式不匹配发生运行时错误。

对于 text_iarchive 来说，编译期存档的输入输出方向 bool 标志 is_loading 和

is_saving 取值与 text_oarchive 正好相反：is_loading::value 的值为 true，而 is_saving::value 的值为 false。

9.4.3 类继承体系

serialization 库的存档类体系与标准流一样复杂，而且用到了模板方式父类调用子类的 CRTP（Curiously Recurring Template Patter）技巧，这里我们以 text_oarchive 为例，它的 UML 类图如图 9-1 所示，便于读者更好地理解存档类的实现：

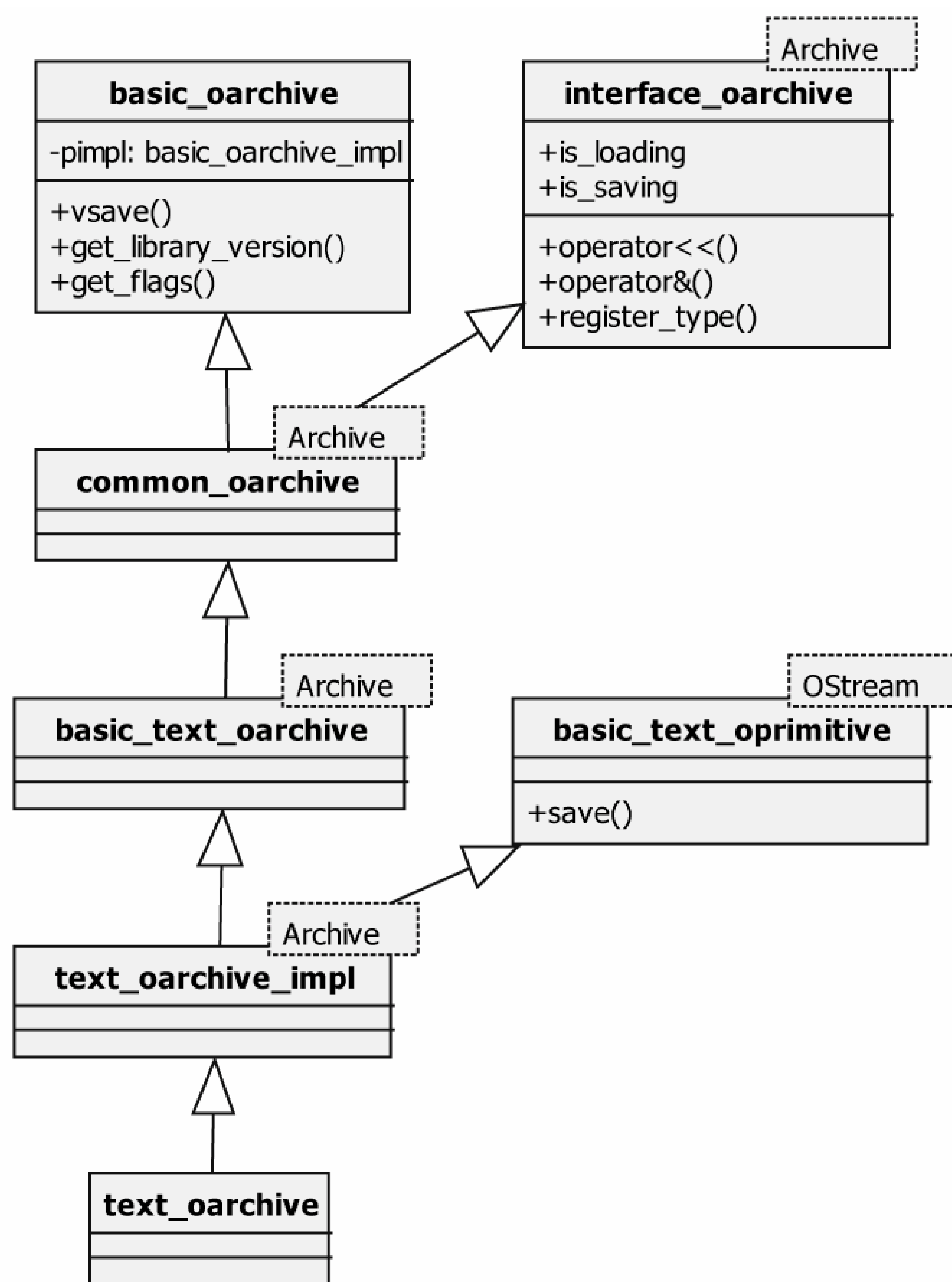


图 9-1 text_oarchive 类图

basic_oarchive 是所有输出存档的基类，它定义了一些可重载的纯虚函数 vsave()，用一个内部类 basic_oarchive_impl 桥接实现了存档必需的核心功能。

interface_oarchive 是一个模板类，它定义了存档的对外接口（如 operator<<），

并使用 CRTP 技巧反向调用子类的成员函数实现了操作符的重载和 register_type()。

common_oarchive 继承了 basic_oarchive 和 interface_oarchive，它利用 interface_oarchive 的 This()函数和 operator<<实现了 basic_oarchive 的 vsave()等接口，把子类真正的实现与虚接口联系在一起。

basic_text_oarchive 提供简单的文本序列化功能，实现了对版本号、类型信息等的序列化。basic_text_oprimitive 以流为模板参数，提供 C++基本数据类型的序列化功能，在析构时自动恢复流的状态。

text_oarchive_impl 继承自 basic_text_oarchive 和 basic_text_oprimitive，实现了文本格式的输出存档，而 text_oarchive 相当于 text_iarchive_impl<text_iarchive>，本身没有任何功能。

9.4.4 XML 格式存档

前面我们使用了文本格式和二进制格式，它们都比较简单，对被序列化的对象不需要什么附加信息，而 XML 格式存档则有些特殊，因为 XML 要求值必须有一个名字（标签）。

为了解决这个问题，serialization 库在名字空间 boost::serialization 里实现了一个辅助函数 make_nvp()，它可以返回一个名字—值对（name-value pair），声明如下：

```
nvp<T> make_nvp(const char * name, T & t);
```

make_nvp()返回的类型 nvp<T>是 std::pair 的子类，它是可序列化的。

xml_oarchive 和 xml_iarchive 必须配合辅助函数 make_nvp()提供正确的名字(标签)才能正常序列化和反序列化，否则会发生错误，例如：

```
#include <boost/archive/xml_iarchive.hpp>     //xml 格式输入存档
#include <boost/archive/xml_oarchive.hpp>     //xml 格式输出存档
...   //省略 archive 名字空间、main 函数、字符串流等代码

using namespace boost::serialization;         //注意名字空间，不是 archive

{
    int i = 10;
    string str("monado");
```

```
    xml_oarchive oa(ss);
    oa << make_nvp("ivalue", i)          //使用标签 ivalue 序列化
       << make_nvp("svalue", str) ;      //使用标签 svalue 序列化
}
{
    int i ;
    string str;

    xml_iarchive ia(ss);
    ia >> make_nvp("i", i)               //使用标签 i 反序列化
       >> make_nvp("s", str) ;           //使用标签 s 反序列化
}
```

序列化形成的 XML 文本大致如下：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="9">
<ivalue>10</ivalue>
<svalue>monado</svalue>
</boost_serialization>
```

使用 make_nvp()指定标签名是一个很麻烦的工作，除了要为对象想出一个独一无二的名字外，我们还必须保证序列化和反序列化时标签名的一致性。为了简化这部分工作，serialization 库利用预处理库 preprocessor 提供了一个很有用的便捷宏 BOOST_SERIALIZATION_NVP，它把对象名转换为标签名，无须我们特别指定，而且序列化和反序列化时的变量名不必相同。

使用 BOOST_SERIALIZATION_NVP 刚才的代码可以简化为：

```
oa << BOOST_SERIALIZATION_NVP(i)
   << BOOST_SERIALIZATION_NVP(str) ;
ia >> BOOST_SERIALIZATION_NVP(i)
   >> BOOST_SERIALIZATION_NVP(str) ;
```

由于 XML 格式存档除了便于阅读外没有更多的优点，所以本书不把它作为讨论的重点。

9.4.5 异常

serialization 库可能抛出的异常 archive_exception 被定义在头文件<boost/archive/archive_exception.hpp>中，类摘要如下：

```
class archive_exception : public virtual std::exception  //虚继承异常类
{
public:
    typedef enum {...} exception_code;                    //错误代码枚举
public:
    exception_code code;
    archive_exception(...);
    ~archive_exception() throw ();
    virtual const char *what( ) const throw();
};
```

archive_exception 是标准异常类 std::exception 的子类，可以用 what() 获得错误信息，也可以直接用成员变量 code 获得错误代码。

9.5 使用序列化

在了解了存档概念后，我们来看看它的用法，看看 serialization 库都提供了哪些序列化的功能。

为了使代码简单清晰起见，接下来我们使用文本格式存档 text_oarchive/text_iarchive 和 std::stringstream 作为示范，并在文件头忽略<boost/archive/text_iarchive.hpp>和<boost/archive/text_oarchive.hpp>的#include 语句。

9.5.1 基本类型的序列化

serialization 库支持对 C++所有基本类型——包括 int、double、bool 和 string 的序列化，可以向存档任意写入这些类型的数据，然后再无损地恢复回来。

示范存档序列化基本类型的代码如下：

```
...   //省略 archive 名字空间、main 函数、字符串流等代码
{
    char c = 'x';               //一些 C++基本类型用于序列化
    int i = 10;
    double d = 2.718;
    bool b = false;
    string s = "metroid";

    text_oarchive oa(ss);       //输出存档，使用已有的字符串流
```

```
    oa << c << i;               //使用 operator<<连续序列化
    oa & d & b & s;             //使用 operator&连续序列化
}

{
    char c ;                    //一些 C++基本类型用于反序列化
    int i ;
    double d ;
    bool b ;
    string s ;

    text_iarchive ia(ss);       //输入存档
    ia & c & i ;                //使用 operator&连续反序列化
    ia >> d >> b >> s;          //使用 operator>>连续反序列化
}
```

存档的序列化和反序列化用法就像是操纵输入输出流，用起来非常简单而且熟悉，被序列化的数据要以同样的类型和顺序反序列化才能正确恢复。需要注意的是因为操作符函数的参数是 T&，所以我们不能使用右值序列化。

operator<<、operator>>和 operator&返回存档自身的引用，所以我们可以像流一样串联操作，在一行代码中连续序列化多个对象，看起来更加简洁。但因为操作符优先级的原因，&和>>、<<混用需要小心，如果必要需使用括号：

```
ia & c >> i ;                   //编译错误
(ia & c) >> i ;                 //正常
ia >> d >> b & s;               //正常
```

虽然&和>>、<<可以混用，但可能会给代码阅读者以混乱的感觉，最好不要这样做。

serialization 库支持枚举类型，它可以看做是整数，所以它的序列化与基本类型没有差别，这里就不举例了，另可参见 9.6.4 小节的序列化 tribool 例子。

9.5.2 数组的序列化

对于 C++原生数组，只要数组里的元素是可序列化的，那么整个数组也是可序列化的：

```
...   //省略 archive 名字空间、main 函数、字符串流等代码
{
    int a[10] = {1,2,3};        //一个可序列化的数组
    text_oarchive oa(ss);
```

```
        oa << a ;                   //序列化数组
    }

    {
        int a[20];                  //用于反序列化的数组，必须足够大
        text_iarchive ia(ss);
        ia >> a;                    //反序列化
    }
```

虽然数组包含长度的信息，序列化时也确实保存了数组的长度，但在反序列化时我们却无法把长度信息恢复到数组中，只能保证有足够大的内存空间供存放数据，否则会导致运行时错误。例如，下面的代码会发生异常：

```
int a[1];                           //用于反序列化的数组，小于实际的长度
ia >> a;                            //反序列化，抛出异常
```

对于 C 字符串（C-Str），archive 会把它视为 char 数组而不是标准字符串（std::string）执行序列化和反序列化：

```
{
    text_oarchive oa(ss);
    oa << "char array";             //序列化一个字符数组，不是字符串
}

{
    text_iarchive ia(ss);
    char buf[20];
    ia >> buf;                      //反序列化得到一个字符数组
}
```

如果我们使用 string 获得反序列化的结果将会得到一串奇怪的数字：

```
string s;
ia >> s;                            //使用 string 反序列化得到错误的结果
```

存档类的 save_binary()和 load_binary()专门用于序列化和反序列化连续的任意内存数据，它可以在一定程度上代替数组的序列化，但使用时需要注意它必须以字节为单位：

```
{
   int a[10] = {1,2,3};             //一个可序列化的数组
    text_oarchive oa(ss);
```

```
    oa.save_binary(a, sizeof(a)) ;         //序列化连续的字节
}

{
    int a[20];                             //用于反序列化的数组，必须足够大
    text_iarchive ia(ss);
    ia.load_binary(a, sizeof(int)* 10);    //反序列化连续的字节
}
```

save_binary()用法比较简单，对于数组和 POD 类型可以直接用 sizeof 操作符计算得到字节大小，而 load_binary()在使用时则必须精确提供被序列化数据的原始大小，否则就会反序列化不完整或者发生异常：

```
ia.load_binary(a, sizeof(int)* 8);         //只反序列化了 8 个整数
ia.load_binary(a, sizeof(int)* 12);        //反序列化了 12 个整数，超长，抛出异常
```

9.5.3 标准类型的序列化

serialization 库支持标准库里定义的 complex、bitset、valarray 和 pair 等类型的序列化。

要序列化这些标准类型，除了要包含基本的存档头文件之外，还要包含它们对应的序列化实现头文件，它们位于<boost/serialization/>内，名字与类型名相同，因而很容易使用，例如 complex 序列化对应的头文件就是<boost/serialization/complex.hpp>。因为这些序列化头文件已经包含了标准类型，因此我们无须特意再包含原头文件（当然重复包含也不算错）。

标准类型与基本类型的序列化方法相同，示范代码如下：

```
#include <boost/serialization/complex.hpp>    //提供复数的序列化能力
#include <boost/serialization/bitset.hpp>     //提供 bitset 的序列化能力
...   //省略 archive 名字空间、main 函数、字符串流等代码

{
  complex<double> c(1, 0);                   //复数类型
  bitset<3> b(BOOST_BINARY(101));            //bitset 类型
  pair<int, int> p(1,2);                     //pair 类型，不必使用专门的序列化头文件

  text_oarchive oa(ss);
```

```
    oa << c << b << p;                          //序列化
  }

  {
    complex<double> c;
    bitset<3> b;
    pair<int, int> p;

    text_iarchive ia(ss);
    ia >> c >> b >> p;                          //反序列化
  }
```

9.5.4 标准容器的序列化

serialization 库对标准容器提供了较完善的支持，包括 vector、deque、list、set/multi_set 和 map/multi_map，但不支持容器适配器 stack、queue 和 priority_queue。要序列化这些标准类型，除了要包含基本的存档头文件之外，同样还要包含容器对应的序列化实现头文件。

示范标准容器序列化的代码如下：

```
#include <boost/serialization/vector.hpp>     //提供 vector 序列化能力
#include <boost/serialization/map.hpp>        //提供 map 序列化能力
...   //省略 archive 名字空间、main 函数、字符串流等代码

   {
      //用 boost.assign 库初始化标准容器
      vector<int> v = list_of(1)(3)(5);
      map<int, string> m = map_list_of(1, "one")(2, "two");

      text_oarchive oa(ss);
      oa << v << m;                             //序列化标准容器
   }

   {
      vector<int> v ;
      map<int, string> m ;

      text_iarchive ia(ss);
      ia >> v >> m;                             //反序列化标准容器
   }
```

9.5.5 非标准容器的序列化

serialization 库支持 SGISTL、STLport 等标准库实现的单链表容器 slist 和非标准散列容器 hash_set、hash_map 的序列化，需要包含对应的序列化实现头文件 <boost/serialization/slist.hpp>、<boost/serialization/hash_set.hpp> 和<boost/serialization/hash_map.hpp>。

这些非标准容器的序列化用法与标准容器相同，这里我们以非标准散列容器 hash_set 和 hash_map 为例，示范用法如下：

```
#include <hash_set>                              //非标准散列集合
#include <hash_map>                              //非标准散列映射

//STLport 需要额外包含 stack_constructor.hpp 才能正常工作
#include <boost/serialization/detail/stack_constructor.hpp>
#include <boost/serialization/hash_set.hpp>
#include <boost/serialization/hash_map.hpp>
...   //省略 archive 名字空间、main 函数、字符串流等代码

{
    hash_set<int> hs = list_of(1)(3)(5);      //散列集合
    hash_map<int, int> hm = map_list_of(1, 2)(3, 4);      //散列映射

    text_oarchive oa(ss);
    oa << hs << hm;                          //序列化
}

{
    hash_set<int> hs ;
    hash_map<int, int> hm;

    text_iarchive ia(ss);
    ia >> hs >> hm;                          //反序列化

    assert(hs.size() == 3);
    assert(hm.size() == 2 && hm[1] == 2);
}
```

至于 VC8 自带的 Dinkumware STL v405，虽然也提供了非标准散列容器 hash_set 和 hash_map，但它们位于 stdext 而不是 std 名字空间里，而且定义也不符合标准（模板

参数只有三个，少一个相等比较谓词），所以不能被序列化。

9.5.6 Boost 类型的序列化

目前 serialization 库支持 Boost 中 date_time、optional、uuid 等少数常用组件的序列化，很遗憾暂时不支持 tribool、rational、tuple 等其他重要类型，不过我们完全可以依据可序列化的概念自行编写代码为它们提供序列化的能力，详见 9.6.4 小节。

date_time 的序列化

要序列化 date_time 组件，首先要编译 date_time 库，并包含头文件 <boost/date_time/gregorian/greg_serialize.hpp> 和 <boost/date_time/posix_time/time_serialize.hpp>，用法如下①：

```
#define BOOST_DATE_TIME_SOURCE
#include <boost/date_time/gregorian/greg_serialize.hpp>
#include <boost/date_time/posix_time/time_serialize.hpp>
... //省略 archive 名字空间、main 函数、字符串流等代码

using namespace boost::gregorian;           //日期名字空间
using namespace boost::posix_time;          //时间名字空间

{
    date d(2011, 5, 1);                     //日期对象
    date_duration dd(10);                   //日期长度对象
    ptime pt(d, hours(8) + minutes(30));    //时间点对象

    text_oarchive oa(ss);
    oa << d << dd << pt;                    //序列化
}

{
    date d;
    date_duration dd;
    ptime pt;

    text_iarchive ia(ss);
```

① Boost1.47 版中 date_time 库新增了一个 cpp 文件<libs/date_time/src/posix_time/posix_time_types.cpp>，但其内容为空，故对嵌入工程编译没有影响。

```
    ia >> d >> dd >> pt;                      //反序列化

    //验证反序列化的正确性
    assert(d.year() == 2011 && d.month() == 5);
    assert(dd == days(10));
    assert(pt.date() == d && pt.time_of_day().hours() == 8);
}
```

optional 的序列化

要序列化 optional 组件，需要包含头文件<boost/serialization/optional.hpp>，用法如下：

```
#include <boost/serialization/optional.hpp>  //提供 optional 序列化能力
...   //省略 archive 名字空间、main 函数、字符串流等代码

{
    optional<int> op(10);                     //一个 optional 对象

    text_oarchive oa(ss);
    oa << op ;                                //序列化
}

{
    optional<int> op;

    text_iarchive ia(ss);
    ia >> op ;                                //反序列化 optional 对象

    assert(op && *op == 10);                  //optional 有值
}
```

uuid 的序列化

要序列化 uuid 组件，需要包含头文件<boost/uuid/uuid_io.hpp>和<boost/uuid/uuid_serialize.hpp>，用法如下：

```
#include <boost/uuid/uuid_generators.hpp>    //uuid 随机生成器
#include <boost/uuid/uuid_io.hpp>            //序列化需使用 uuid 的 io 能力
#include <boost/uuid/uuid_serialize.hpp>     //提供 uuid 的序列化能力
```

```
...    //省略 archive 名字空间、main 函数、字符串流等代码

{
    uuids::uuid u = uuids::random_generator()();      //一个随机 uuid

    text_oarchive oa(ss);
    oa << u ;                                          //序列化 uuid
}

{
    uuids::uuid u;

    text_iarchive ia(ss);
    ia >> u;                                           //反序列化 uuid

    assert(u.version() ==
          uuids::uuid::version_random_number_based);
}
```

9.5.7 Boost 容器的序列化

serialization 库对 Boost 的容器提供了有限的支持，包括指针容器、多索引容器、array 和 variant，而侵入式容器因为它不能算是真正的“容器”，所以不能被序列化。

指针容器的序列化

所有的指针容器都是可序列化的，容器的序列化头文件位于目录<boost/ptr_container/>下，每一种容器的序列化头文件形如 serialize_ptr_xxx.hpp。

因为指针的序列化比较特殊，所以这部分内容放在 9.8.5 小节介绍。

多索引容器的序列化

多索引容器库 multi_index 自身就支持序列化和反序列化，不需要包含特殊的头文件，当多索引容器被反序列化恢复时所有的元素对应的索引顺序也被同时恢复。

示范多索引容器序列化的代码如下：

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/ordered_index.hpp>        //有序索引
#include <boost/multi_index/hashed_index.hpp>         //散列(无序)索引
```

```
#include <boost/multi_index/key_extractors.hpp>  //键提取器
...  //省略 archive 名字空间、main 函数、字符串流等代码

typedef multi_index_container<int,
    indexed_by<
        ordered_unique<mi::identity<int>>,
        hashed_unique<mi::identity<int>>
        >
    > mic_t;                                    //多索引容器定义，两个简单的索引

{
    mic_t mic;                                  //一个多索引容器
    insert(mic)(1), 3, 5;

    text_oarchive oa(ss);
    oa << mic;                                  //序列化
}

{
    mic_t mic;

    text_iarchive ia(ss);
    ia >> mic ;                                 //反序列化

    assert(mic.size() == 3);
    assert(mic.get<1>().count(5) == 1);
}
```

其他容器的序列化

serialization 库还提供对 array 和 variant 的序列化支持，需要包含头文件<boost/serialization/array.hpp>或<boost/serialization/variant.hpp>，示范代码如下：

```
#include <boost/serialization/array.hpp>
#include <boost/serialization/variant.hpp>
...  //省略 archive 名字空间、main 函数、字符串流等代码
{
    array<int, 5> ar = {0,1,2,3,4};             //数组容器
    variant<int, string> var("pikimin");        //泛型联合容器
```

```
    text_oarchive oa(ss);
    oa << ar << var;                        //序列化
}

{
    array<int, 5> ar;
    variant<int, string> var;

    text_iarchive ia(ss);
    ia >> ar >> var ;                       //反序列化

    assert(ar[0] == 0 && ar[4] == 4);
    assert(var.which() == 1 &&
        get<string>(var) == "pikimin");
}
```

9.6 定制序列化

serialization 库已经为已经存在的大量类型提供了序列化的能力，在这些已有类型的基础上，我们可以依据可序列化的定义（参见 9.3.2 小节）定制实现自定义类型的可序列化。存档将调用类相关的 serialize()函数递归执行，直至完成所有数据的序列化和反序列化。

9.6.1 可序列化的要求

自定义类型可以使用两种方式实现可序列化：侵入式和非侵入式。

侵入式可序列化

侵入式的可序列化要求类必须有如下形式的一个成员函数 serialize()：

```
template<class Archive>   //存档模板，可以是输入存档或者输出存档
void serialize(Archive &ar, const unsigned int version)
{
    ...                      //基于已有类型的序列化代码
}
```

serialize()有两个参数，第一个参数是被用于序列化或反序列化的存档，第二个参

数是一个整型的版本号，它标记了自定义类序列化的版本，将在 9.7.2 小节详述，现在可以暂时忽略。

serialize()是一个模板函数，这意味着它可以泛型地接受输入存档或者输出存档，因为这两种存档都重载了 operator&，所以我们可以无须判断存档类型，“多态”地使用相同的代码来执行序列化和反序列化操作，简化了代码的编写。

在 serialize()函数内部，我们应该使用 operator&存取类的所有必要数据成员，有点儿像是向文件中写入数据，顺序和格式都可以自己定，但必须保证对象可以在序列化时存储了足够的信息，在反序列化时才能够完全恢复对象的状态。

boost::serialization::access 是一个辅助类，声明了一系列的静态成员函数间接调用自定义类的 serialize()，存档通过它来完成对自定义类的序列化（类似 3.4 节介绍的辅助类 iterator_core_access）。为了让 access 可以调用 serialize()，我们需要使用友元的方式授予访问权限：

```
friend class boost::serialization::access;
```

使用辅助类 access 后，serialize()最好被设为 private，表示仅供类本身使用[①]。

非侵入式可序列化

有的时候修改类定义很困难甚至是不可能的（比如标准库里的所有类型），这时我们就只能使用非侵入的方式，定义一个如下形式的自由函数 serialize()：

```
template<class Archive>
inline void serialize( Archive & ar, some_class & t,
                      const unsigned int file_version)
{
    ...    //基于已有类型的序列化代码
}
```

自由函数 serialize()与成员函数 serialize()很相似，只是多了一个自定义类型的参数 t，在函数体中我们使用它来完成序列化和反序列化操作。因为非侵入式不能访问类的私有成员，所以要求自定义类必须对外提供足够多的细节才能够正确序列化和反序列化。

为了方便编译器查找自由函数 serialize()，通常它应该位于名字空间 boost::serialization，或者是 boost::archive 和自定义类型所在的名字空间。

① 当然把 serialize()直接声明为 public 也可以，但这种做法不好，因为序列化功能通常不应该直接被外部调用。

分解序列化和反序列化

serialize()函数是存档实际调用的函数，也是我们通常的最佳选择，因为序列化和反序列化的执行顺序被保证是一致的，减少了可能发生的错误。但也有可能我们想要不同的序列化和反序列化代码，比如根据版本号或者某些数据成员执行不同的分支，这时我们可以把 serialize()函数分解为 save()和 load()两个函数。

要分解成员函数 serialize()，需要包含头文件<boost/serialization/split_member.hpp>，实现 save()和 load()如下：

```
template<class Archive>
void save(Archive & ar, const unsigned int version) const;
template<class Archive>
void load(Archive & ar, const unsigned int version);
```

然后我们需要使用 boost::serialization::split_member()函数来把 save()和 load()组合成实际的成员函数 serialize()，像这样：

```
template<class Archive>
void serialize(Archive & ar, const unsigned int version )
{
    boost::serialization::split_member(ar, *this, version);
}
```

要分解自由函数 serialize()，需要包含头文件<boost/serialization/split_free.hpp>，实现 save()和 load()如下：

```
template<class Archive>
void save(Archive & ar, const some_class & t, const unsigned int version);
template<class Archive>
void load(Archive & ar, some_class & t, const unsigned int version);
```

这两个自由函数对应的组合函数是 split_free()：

```
void serialize(Archive & ar, some_class & t,const unsigned int version)
{
    boost::serialization::split_free(ar, t, version);
}
```

serialization 库额外提供了两个简易宏 BOOST_SERIALIZATION_SPLIT_MEMBER()

和 BOOST_SERIALIZATION_SPLIT_FREE(T)，用来简化 serialize()函数的编写，但它们仅适用于很少的简单情形，大多数时候我们还是需要手工实现。

9.6.2 侵入式可序列化

我们使用第 7 章定义的 person 类作为例子，演示侵入式可序列化的使用，在此把 person 类声明如下：

```
class person
{
public:
  int m_id;                                          //身份标识号
  string m_fname, m_lname;                           //姓名

  person(){}                                         //增加一个缺省构造函数
  person(int id, const string& f, const string& l):  //构造函数
  m_id(id), m_fname(f), m_lname(l){}
};
```

现在 person 类不是可序列化的，因为它不满足可序列化的要求。我们必须为 person 类实现成员函数 serialize()，使用 operator&操作其数据成员，并声明友元类 boost::serialization::access：

```
class person
{
  ... //前略
private:
  friend boost::serialization::access;               //声明友元，授予访问权限

  template<typename Archive>
  void serialize(Archive & ar,const unsigned int version)//序列化函数
  {
      ar & m_id;                  //依据存档类型序列化或反序列化 id
      ar & m_fname & m_lname;     //依据存档类型序列化或反序列化名字
  }
};
```

相当简单，只需要寥寥几行代码我们就为 person 类添加了可序列化能力，现在它就可以被存档类任意存取了，示范代码如下：

```
int main()
```

```
{
  stringstream ss;

  person p1(1, "anderson", "neo");        //被序列化的对象
  text_oarchive(ss) << p1;                //序列化

  person p2;                              //被反序列化的对象
  text_iarchive(ss) >> p2;                //反序列化

  assert(p1 == p2);                       //两者等价
}
```

person 是可序列化的，所以包含它的数组、容器和数据结构也都是可序列化的：

```
deque<person> dq1;                        //双端队列，可序列化
push_back(dq1)(1, "anderson", "neo")()();//用 assign 库添加数据
text_oarchive(ss) << dq1;                 //序列化

deque<person> dq2;                        //被反序列化的对象
text_iarchive(ss) >> dq2;                 //反序列化

assert(dq1 == dq2);                       //两个队列等价
```

9.6.3 非侵入式可序列化

person 类公开了内部的数据成员，暴露了所有细节，因此我们也可以使用非侵入方式实现它的可序列化。

这里我们在名字空间 boost::serialization 里实现自由函数 serialize()，代码与侵入式差不多：

```
namespace boost {                         // namespace boost
namespace serialization {                 // namespace serialization

template<class Archive>
void serialize(Archive & ar, person & p, const unsigned int version)
{
  ar & p.m_id;                            //依据存档类型序列化或反序列化 id
  ar & p.m_fname & p.m_lname;             //依据存档类型序列化或反序列化名字
}
```

```
} // namespace serialization
} // namespace boost
```

如果要分离序列化和反序列化代码，我们需要编写 save() 和 load()，代码如下[1]：

```
#include <boost/serialization/split_free.hpp>    //分离必要的头文件

template<class Archive>
void save(Archive & ar, const person & p, const unsigned int version)
{
  cout << "call save" << endl;
  ar << p.m_id;
  ar << p.m_fname << p.m_lname;
}
template<class Archive>
void load(Archive & ar, person & p, const unsigned int version)
{
  cout << "call load" << endl;
  ar >> p.m_id;
  ar >> p.m_fname >> p.m_lname;
}
BOOST_SERIALIZATION_SPLIT_FREE(person)            //组合分离的函数
```

分离序列化和反序列化代码后 person 的可序列化能力不变。

9.6.4 Boost 类型的可序列化

使用非侵入式可序列化的方式，我们就可以为 Boost 库中的大部分组件自行添加序列化的支持。因为这些组件大多数是模板类，因此如果要分离序列化和反序列化代码就不能使用宏 BOOST_SERIALIZATION_SPLIT_FREE。

tribool 的可序列化

首先来看 tribool 的序列化，它是一个非模板类，而且有 public 成员 value 保存了三态 bool 的枚举值，所以它的可序列化实现非常简单：

```
#include <boost/logic/tribool.hpp>    //tribool 头文件
```

① 使用 BOOST_SERIALIZATION_SPLIT_FREE 宏时偶尔可能会遇到名字空间解析错误，这时可尝试改变 save() 和 load() 所在的名字空间，例如从 boost::serialization 变更到自定义类所在的名字空间。

```
namespace boost {          // namespace boost
namespace logic {          // 使用 boost::logic 名字空间，也可以用 serialization

template<class Archive>   //输出到存档
void save(Archive & ar, const tribool & tb, const unsigned int)
{
  ar << tb.value;
}
template<class Archive>   //从存档输入
void load(Archive & ar, tribool & tb, const unsigned int)
{
  ar >> tb.value;
}

template<class Archive>   //组合 save()和 load()函数
void serialize(Archive & ar, tribool & tb, const unsigned int version)
{
  boost::serialization::split_free(ar, tb, version);
}

}                          // namespace logic
}                          // namespace boost
```

注意：因为我们使用的名字空间是 boost::logic，所以不能使用宏 BOOST_SERIALIZATION_SPLIT_FREE，原因是宏内部没有使用名字空间 boost::serialization 来限定 split_free()，使用的话会发生找不到标志符编译错误。

由于 tribool 类型实在是太简单了，所以实际上我们不必使用分离的方式，完全可以直接实现 serialize()函数。

```
template<class Archive>
void serialize(Archive & ar, tribool & tb, const unsigned int version)
{
  ar & tb.value;           //直接序列化或反序列化 tribool 的枚举值
}
```

验证 tribool 可序列化的代码如下：

```
tribool tb1 = true;        //indeterminate;
text_oarchive(ss) << tb1;//序列化
```

```
  tribool tb2;
  text_iarchive(ss) >> tb2;         //反序列化

  assert(!indeterminate(tb2));
  cout << (tb2 == true) << endl;    //输出 1，即 true
```

rational 的可序列化

rational 是一个模板类，实现了有理数的 C++表述，它提供了成员函数 numerator()和 denominator()访问分子和分母，成员函数 assign()可以创建一个有理数。使用这几个接口我们就能够实现 rational 的可序列化：

```
namespace boost {                   // namespace boost
namespace serialization {           // namespace serialization

template<class Archive, typename I>  //I 是 rational 的模板参数
void save(Archive & ar, const rational<I> & r, const unsigned int)
{
  I n = r.numerator();              //获得分子
  I d = r.denominator();            //获得分母
  ar << n << d;                     //顺序序列化
}
template<class Archive, typename I>  //I 是 rational 的模板参数
void load(Archive & ar, rational<I> & r, const unsigned int)
{
  I n, d;                           //用于恢复分子和分母
  ar >> n >> d;                     //顺序反序列化
  r.assign(n, d);                   //赋值恢复有理数
}

template<class Archive, typename I>  //组合 save()和 load()函数
void serialize(Archive & ar, rational<I> & r, const unsigned int version)
{
  boost::serialization::split_free(ar, r, version);
}
} // namespace serialization
} // namespace boost
```

在这里我们需要注意的是因为 numerator()和 denominator()返回的是右值，所以不能直接用 operator<<序列化，必须先使用临时变量暂存才能序列化。

验证 rational 可序列化的代码如下：

```
rational<int> r1(22, 7);  //一个有理数对象;
text_oarchive(ss) << r1;  //序列化

rational<int> r2;
text_iarchive(ss) >> r2;  //反序列化

assert(r1 == r2);         //验证反序列化的正确性
```

读者可仿照本节的例子，再参考 serialization 库的 complex、optional 序列化头文件实现对其他类型的可序列化。

9.6.5 Boost 容器的可序列化

同样使用非侵入式可序列化的方式，参考 vector、array 等的实现我们可以为 Boost 库中的其他容器自行添加序列化的支持。

由于这些容器都与标准容器具有互操作性，所以我们可以使用 vector 来作为数据的临时存储，利用 vector 的可序列化能力来方便地实现 Boost 容器的可序列化，接下来本书以 unordered 和 multi_array 作为示范的例子，其他容器读者可仿照实现。

unordered 的可序列化

unordered 组件实现了 tr1 定义的散列容器，这里我们仅实现 unordered_set 可序列化，由于它有四个模板参数，所以 save()、load()和 serialize()函数的模板参数列表也要修改一致：

```
namespace boost {                         // namespace boost
namespace serialization {                 // namespace serialization

//unordered_set 需要有 4 个模板参数
template<class Archive, typename T, typename H, typename P, typename A>
void save(Archive & ar,
  const unordered_set<T, H, P, A> & s, const unsigned int)
{
  vector<T> vec(s.begin(), s.end());      //复制 unordered 的元素到 vector
  ar << vec;                              //序列化
}
```

```
template<class Archive, typename T, typename H, typename P, typename A>
void load(Archive & ar,
  unordered_set<T, H, P, A> & s, const unsigned int)
{
  vector<T> vec;                          //用于恢复数据
  ar >> vec;                              //反序列化数据到 vector
  std::copy(vec.begin(), vec.end(),       //unordered_set 没有 assign 函数
      std::inserter(s, s.begin()));       //所以使用 copy 算法恢复数据
}

//组合 save()和 load()函数
template<class Archive, typename T, typename H, typename P, typename A>
void serialize(Archive & ar,
  unordered_set<T, H, P, A> & s, const unsigned int version)
{
  boost::serialization::split_free(ar, s, version);
}

}                                         // namespace serialization
}                                         // namespace boost
```

验证 unordered_set 可序列化的代码如下：

```
unordered_set<int> s1  = list_of(1)(2)(3);  //使用 assign 库添加数据
text_oarchive(ss) << s1;                     //序列化

unordered_set<int> s2;
text_iarchive(ss) >> s2;                     //反序列化

assert(s1 == s2);                            //验证反序列化的正确性
```

multi_array 的可序列化

multi_array 实现了多维数组，它可以用成员函数 data()和 num_elements()获得容器内元素的数组地址和数量，所以实现也并不困难：

```
namespace boost {                         // namespace boost
namespace serialization {                 // namespace serialization

template<class Archive, typename T, std::size_t N>
void save(Archive & ar,
```

```
    const multi_array<T, N> & ma, const unsigned int)
{
    vector<T> vec(ma.data(), ma.data() + ma.num_elements());
    ar << vec;                          //序列化
}

template<class Archive, typename T, std::size_t N>
void load(Archive & ar,
    multi_array<T, N> & ma, const unsigned int )
{
    vector<T> vec;                      //用于恢复数据
    ar >> vec;                          //反序列化到 vector

    ma.assign(vec.begin(), vec.end());  //使用 assign 赋值恢复
}

//组合 save()和 load()函数
template<class Archive, typename T, std::size_t N>
void serialize(Archive & ar,
    multi_array<T, N> & ma, const unsigned int version)
{
    boost::serialization::split_free(ar, ma, version);
}

} // namespace serialization
} // namespace boost
```

验证 multi_array 可序列化的代码如下：

```
multi_array<int, 2> ma1(extents[2][2]);  //2*2 多维数组
ma1[0][0] = 100;
ma1[1][1] = 200;
text_oarchive(ss) << ma1;                //序列化

multi_array<int, 2> ma2(extents[2][2]);
text_iarchive(ss) >> ma2;                //反序列化

assert(ma1 == ma2);                      //验证反序列化的正确性
assert(ma2[1][1] == 200);
```

9.7 高级定制序列化

本节我们将讨论关于自定义类序列化的一些更深入的议题。

9.7.1 派生类的序列化

在一个类继承体系中每个类都可以是可序列化的，各个层次的类需要分别声明 friend boost::serialization::access 并实现成员函数 serialize()，不过为了使基类的序列化功能被正确调用，需要在 serialize() 中增加一条额外的调用语句：

```
ar & boost::serialization::base_object<base_class>(*this);
```

这里的 base_object() 是 serialization 库的一个辅助函数，专门用于派生类访问基类，它的简要声明如下：

```
template<class Base, class Derived>
Base& base_object(Derived &d);
```

base_object() 有两个模板参数，分别是基类和派生类，由于派生类 Derived 可以由函数参数推导出来，所以我们只需要指定基类类型就可以了。base_object() 功能上类似：

```
ar & static_cast<base_class>(*this);
```

但它可以保证多态类正确地序列化，不会损失数据。

下面我们定义一个派生自 person 的新类 worker，它新增了年龄和职业两个成员变量：

```
class worker: public person                          //从 person 派生
{
public:
  int age;                                           //年龄
  string job;                                        //职业

  worker():person(){}                                //构造函数
  worker(int id, const string& f, const string& l,   //构造函数
      int a, const string& j):
      person(id,f,l), age(a), job(j){}
```

```
private:
  friend boost::serialization::access;                //声明友元，授予访问权限
  template<typename Archive>
  void serialize(Archive & ar, const unsigned int version)
  {
      ar & boost::serialization::base_object<person>(*this);
      ar & age;                                       //序列化年龄
  }
};
```

现在 worker 就可以正确地被序列化了：

```
ofstream ofs("serial.txt");                           //输出文件流
worker p1(1, "anderson", "neo", 25, "engineer");      //一个 worker 对象
text_oarchive(ofs) << p1;                             //序列化

ifstream ifs("serial.txt");                           //输入文件流
worker p2;
text_iarchive(ifs) >> p2;                             //反序列化

assert(p2.age == p1.age);                             //验证反序列化的正确性
```

9.7.2 序列化的版本

对象的序列化经常会遇到版本兼容的问题，通常情况是新版本的类持有更多的属性，但在序列化和反序列化时还必须兼容以前版本的数据。

serialization 库对此给出的解决方案是使用“版本号”，也就是 serialize() 函数的第二个参数。存档在序列化保存数据时会自动记录当前类的版本号，反序列化时也会自动读出版本号，类可根据版本号决定序列化的策略。

我们可以使用宏 BOOST_CLASS_VERSION(T, N) 为可序列化类型 T 指定版本，其中 N 是版本号。宏被展开为一个 version<T>值元函数的特化形式，由于使用了模板元编程，它的最大值不能超过 255。如果不指定版本号（我们之前的所有示例都是这样），那么缺省值是 0。

以 9.7.1 小节的 worker 类作为例子，第 0 版只序列化了 age 成员，序列化的数据是：

```
22 serialization::archive 9 0 0 0 0 1 8 anderson 3 neo 25
```

现在第 1 版打算把职业也保存起来，为了兼容旧版数据需要做如下的修改：

```
class worker: public person
{
  ... //同前
private:
  friend boost::serialization::access;
  template<typename Archive>
  void serialize(Archive & ar, const unsigned int version)
  {
      ar & boost::serialization::base_object<person>(*this);
      ar & age ;
      if (version > 0)           //根据版本号做分支处理
      {   ar & job;}             //版本号大于 0 则存取 job 成员
  }
};
BOOST_CLASS_VERSION(worker, 1)   //定义序列化的版本号为 1，也可以是任意数字
```

这样我们就可以轻松兼容旧版本数据了，代码如下：

```
ifstream ifs("serial.txt");      //打开旧版本的数据文件
worker p1;
text_iarchive(ifs) >> p1;        //反序列化

assert(p1.job.empty());          //旧版本数据不包含职业信息

p1.job = "engineer";             //添加职业信息

ofstream ofs("serial1.txt");     //保存新版本数据
text_oarchive(ofs) << p1;        //序列化

ifstream ifs1("serial1.txt");    //读取新版本数据
worker p2;
text_iarchive(ifs1) >> p2;       //反序列化

assert(p2.job == "engineer");    //验证新版本数据
```

新版本的序列化数据记录了版本号和新增的 job 信息：

```
22 serialization::archive 9 0 1 0 0 1 8 anderson 3 neo 25 8 engineer
```

9.8　指针的序列化

指针是 C++中一种比较特殊的类型，所以 serialization 库对它的序列化和反序列化有特殊的处理，并不是序列化指针的值（对象的地址）而是序列化指针指向的内容。

9.8.1　指针可序列化的要求

serialization 库不支持对基本类型的指针的序列化，下面的代码会编译报错：

```
int *p = new int(10);          //整型的指针
string *s = new string;        //字符串指针
text_oarchive(ss) << p;        //编译失败
text_oarchive(ss) << s;        //编译失败
```

serialization 库只能序列化和反序列化有 serialize()的类的类型指针，也就是说标准库和 Boost 库里的大部分类型以及自定义类型，序列化时从指针获得对象，反序列化时自动分配内存创建新对象。

因为反序列化指针需要分配内存，所以对于指针的可序列化类型还有一个额外的要求，类型必须有缺省或无参构造函数，这样存档才可以用形如 new T()的代码来初始化内存。如果类没有缺省构造函数，那么我们需要在名字空间 boost::serialization 编写一个 load_construct_data()函数调用构造函数初始化对象。

例如，如果person类没有缺省构造函数，那么我们需要编写load_construct_data()函数如下：

```
namespace boost {                  // namespace boost
namespace serialization {          // namespace serialization

template<class Archive>
inline void load_construct_data(
          Archive & ar, person * p, const unsigned int version)
{
  // 调用定位 new 表达式来初始化 person 对象，给出假数据构造即可
  ::new(p)person(0, "fname", "lname");
}
}                                  // namespace serialization
}                                  // namespace boost
```

9.8.2 原始指针的序列化

如果指针指向的是可序列化的类类型，那么它就可以被存档正确地序列化和反序列化。

下面的代码示范了对标准容器 vector 的指针的序列化和反序列化：

```
vector<int> v = list_of(1)(2)(3);//一个 vector 容器，容纳 int

vector<int> *p1 = &v;          //容器的指针，元素 int 不影响 vector 的可序列化
text_oarchive(ss) << p1;       //序列化指针

vector<int> *p2 ;              //用于恢复容器的指针，无须特地分配内存
text_iarchive(ss) >> p2;       //反序列化，自动恢复指针

assert(*p2 == v);              //验证
```

自定义类型的指针也可以序列化和反序列化：

```
person *p1 = new person(1, "anderson", "neo");  //一个自定义类型的指针
text_oarchive(ss) << p1;       //序列化指针

person *p2;                    //用于恢复容器的指针，无须特地分配内存
text_iarchive(ss) >> p2;       //反序列化，自动恢复指针

assert(*p1 == *p2);            //验证
```

9.8.3 智能指针的序列化

C++原始指针很不安全，容易造成很多安全隐患，我们更常用的是各种智能指针，例如标准库的 std::auto_ptr 和 Boost 的 scoped_ptr、shared_ptr 等，serialization 库也对这些智能指针提供了序列化支持。

serialization 库直接支持 scoped_ptr、weak_ptr 和 shared_ptr 的序列化，需要包含头文件<boost/serialization/scoped_ptr.hpp>、<boost/serialization/weak_ptr.hpp>或<boost/serialization/shared_ptr.hpp>，序列化这几种智能指针与原始指针用法一样：

```
#include <boost/serialization/scoped_ptr.hpp>
#include <boost/serialization/shared_ptr.hpp>
```

```
scoped_ptr<person> p1(new person(1, "anderson", "neo"));
shared_ptr<person> sp1(new person(2, "agent", "smith"));
text_oarchive(ss) << p1 << sp1;        //序列化智能指针

scoped_ptr<person> p2;
shared_ptr<person> sp2;
text_iarchive(ss) >> p2 >> sp2;        //反序列化智能指针

assert(*p1 == *p2);                    //验证
assert(*sp1 == *sp2);
```

同样的，我们不能对基本类型的智能指针执行序列化：

```
scoped_ptr<int> p1(new int(10));
shared_ptr<string> sp1(new string("metroid"));
text_oarchive(ss) << p1 << sp1;            //编译错误
```

serialization 库没有直接支持 std::auto_ptr 的序列化，而是作为示范代码在 <libs/serialization/src/demo_auto_ptr.cpp>提供了实现，如果需要序列化 std::auto_ptr 读者可以自行提取其中的代码形成头文件使用。

9.8.4 派生类指针的序列化

对于派生类指针的序列化处理要复杂一些，因为存档无法仅通过基类指针得知实际的类型新型，如果类不是多态的（没有虚函数），那么存档只能认为这个指针指向的是基类，只调用基类的序列化代码，派生类的数据无法序列化或反序列化。

如果像下面这样序列化指针，将无法恢复派生类的信息：

```
person *p1 = new worker(1, "anderson", "neo", 25, "engineer");
text_oarchive oa(ss) << p1;                //只能序列化基类的信息
```

序列化的数据是：

```
22 serialization::archive 9 0 1 0 0 1 8 anderson 3 neo
```

很显然，派生类 worker 的 age 和 job 没有被序列化。

如果想让存档自动识别类的继承体系，那么这些类就必须都是多态的，也就是说类里至少要有一个虚函数，然后使用存档的模板成员函数 register_type<D>()用来向存档“注册”派生类 D 的信息。

把类变为多态类最简单的方法是为它增加虚析构函数：

```
class person
{
  ... //同前
  virtual ~person(){}                //虚析构函数
};

class worker: public person
{
  ... //同前
  virtual ~worker(){}                //虚析构函数
};
```

然后只要在序列化和反序列化之前注册类型信息，存档就能够正确地识别指针的类型：

```
person *p1 = new worker(1, "anderson", "neo", 25, "engineer");

text_oarchive oa(ss);
oa.register_type<worker>();        //向输出存档注册类型
oa << p1 ;                         //多态类的指针可以被正确识别为派生类

person *p2;

text_iarchive ia(ss);
ia.register_type<worker>();        //向输入存档注册类型
ia >> p2 ;                         //反序列化会正确恢复为派生类
```

这种方式虽然可用，但不太方便，如果类的继承体系很庞大的话，那么注册派生类的代码的维护工作将会相当麻烦，所以 serialization 库提供了另外一种简单的方法，可以在类的定义后用一个唯一的字符串标识类型，这样存档就可以根据注册的字符串来确定类型信息，无须每次使用存档前手动注册。

头文件<boost/serialization/export.hpp>包含了两个宏用来简化这个工作：

```
#define BOOST_CLASS_EXPORT_GUID(T, K)
#define BOOST_CLASS_EXPORT(T)
```

宏 BOOST_CLASS_EXPORT_GUID 使用一个唯一字符串 K 来注册类型 T，而宏 BOOST_CLASS_EXPORT 则直接使用类型名作为标识字符串注册，用起来更加简单。为了使宏注册总能成功，头文件<boost/serialization/export.hpp>必须位于所有存档头文件之后。

使用 BOOST_CLASS_EXPORT 可以注册 worker 类如下：

```
class worker: public person
{...};                       //同前
BOOST_CLASS_EXPORT(worker)   //注册多态派生类
//相当于 BOOST_CLASS_EXPORT_GUID(worker, "worker")
```

在实际开发过程中手动注册和自动注册的方式可以混用，读者可自行选择最适合自己的方式。

9.8.5 指针容器的序列化

只要元素不是基本类型而且是可序列化的，那么所有的 Boost 指针容器（见第 5 章）都是可序列化的，我们只需要包含目录<boost/ptr_container/>下对应的序列化头文件。

示范指针容器序列化的代码如下：

```
#include <boost/ptr_container/serialize_ptr_vector.hpp>//序列化支持
#include <boost/assign/ptr_list_inserter.hpp>
...
ptr_vector<person> p1;        //指针向量容器，容纳多态对象

//插入多态对象
ptr_push_back<person>(p1)(1, "anderson", "neo");
ptr_push_back<worker>(p1)(2, "agent", "smith", 30, "engineer");

text_oarchive(ss) << p1;      //序列化指针容器

ptr_vector<person> p2;
text_iarchive(ss) >> p2;      //反序列化指针容器

assert(p2.size() == 2);       //验证反序列化结果
assert(p1 == p2);
```

9.9 实用工具

serialization 库在开发的过程中为了方便实现了许多有用的小工具，其中一部分对我们用户也颇具实用价值，本节将摘要性地介绍其中的几个。

9.9.1 BOOST_STRONG_TYPEDEF

标准的 typedef 关键字可以为类型创建别名，相当于语法层面的宏替换，并不是一个

新类型，例如：

```
typedef int my_int;                          //定义类型 my_int，是 int 的别名
assert(typeid(int) == typeid(my_int));       //两者运行时类型相同
assert((is_same<int, my_int>::value));       //两者编译期类型相同
```

而 serialization 库在头文件<boost/serialization/strong_typedef.hpp>提供了一个类似 typedef 的宏 BOOST_STRONG_TYPEDEF，它可以定义一个与原类型功能相同的新类型。

宏 BOOST_STRONG_TYPEDEF 声明如下：

```
#define BOOST_STRONG_TYPEDEF(T, D)
```

它定义了一个新类型 D,运用了 operators 库的 totally_ordered 操作符重载能力，为类型 D 实现了对 T 的全序运算和隐式转换。D 与 T 两者功能等价，可互换，但类型不同：

```
BOOST_STRONG_TYPEDEF(int, my_int) ;                //my_int 是个新类型
assert(typeid(int) != typeid(my_int));             //两者运行时类型不同
assert((!is_same<int, my_int>::value));            //两者编译期类型不同
assert((is_convertible<my_int, int>::value));      //可隐式转换
```

9.9.2 BOOST_STATIC_WARNING

serialization 库在头文件<boost/serialization/static_warning.hpp>里提供一个编译器警告宏 BOOST_STATIC_WARNING，声明如下：

```
#define BOOST_STATIC_WARNING(B)
```

它与 Boost 的静态断言 BOOST_STATIC_ASSERT 的用法几乎完全相同，只是产生的是编译警告并包括所在的行数，而不是编译错误。

示范 BOOST_STATIC_WARNING 用法的代码如下：

```
BOOST_STATIC_WARNING(sizeof(int)==0);              //产生一个编译警告
```

9.9.3 smart_cast

我们已经在 2.6 小节讨论了转型的问题，serialization 库为了解决泛型语境下识别多态类的问题又提供了一个新的“智能”转型工具 smart_cast，使用模板元编程技术自动选择最合适的转型操作符，它位于名字空间 boost::serialization，需要包含头文件

<boost/serialization/smart_cast.hpp>。

smart_cast 有两个转型函数：

```
template<class T, class U>
T smart_cast(U u);                    //可转型指针和引用
template<class T, class U>
T smart_cast_reference(U & u);        //专门转型引用
```

smart_cast()是最简单易用的转型函数，它可以转型指针和引用，把 U 类型转型为 T 类型，通常我们只需要指明转型的目标类型 T 即可，类型 U 可以自动推导，但如果 T 和 U 都是引用，那么就需要同时写出 T 和 U，或者使用意义更明确的 smart_cast_reference。

示范这两个转型函数用法的代码如下：

```
person p, *pp , &rp = p;                //person 对象，指针和引用
worker w, *pw = &w, &rw = w;            //worker 对象，指针和引用

//转型函数位于名字空间 serialization
using namespace boost::serialization;

pp = smart_cast<worker*>(&w);           //转型指针
rp = smart_cast<person&, worker&>(rw);  //转型引用，指明两个参数
rp = smart_cast_reference<person&>(rw); //转型引用
```

9.9.4 base64 编解码

serialization 库使用 iterators 库（参见第 3 章）实现了 base64 编解码功能，这些迭代器位于名字空间 boost::archive::iterators。

base64 编码需要使用以下三个迭代器类：

- transform_width ：把 x 位的字节序列转换为 y 位的字节序列，对于 base64 编码来说就是把 8 位转换为 6 位；
- base64_from_binary：把原始字节序列转换为 base64 编码；
- binary_from_base64：把 base64 编码转换为原始字节序列。

transform_width

transform_width 使用了迭代器适配器 iterator_adaptor（见 3.5 小节），位于

头文件<boost/archive/iterators/transform_width.hpp>，声明如下：

```
template<
    class Base,                 //被适配的迭代器，通常是字节序列指针
    int BitsOut,                //转换后的位数
    int BitsIn                  //转换前的位数
>
class transform_width : public boost::iterator_adaptor<...>
{...};
```

transform_width 可以把 BitsIn 位的字节序列转换为 BitsOut 的字节序列，示范代码如下：

```
char c1[4] = {1,2,3,4};                              //输入字节序列
vector<char> v;                                      //输出字节序列

using namespace boost::archive::iterators;           //名字空间
typedef transform_width<char*, 6, 8 > tw68;          //迭代器类型定义
std::copy(                                           //copy 算法
    tw68(c1), tw68(c1 + 3),                          //转换 3 个字节
    back_inserter(v));                   //输出 4 个字节：0x00,0x10,0x08,0x03
```

需要注意的是如果输入长度不是 BitsOut 的整数倍，transform_width 会越界读取数据，这会造成意料不到的结果。比如，如果我们转换 4 个字节，由于 4 个字节是 32 位，不是 6 的整数倍，所以最后的 2 位为了补齐会多读 4 位，代码如下：

```
std::copy(                    //copy 算法
  tw68(c1), tw68(c1 + 4),     //转换 4 个字节
  back_inserter(v));          //输出 6 个字节：0x00,0x10,0x08,0x03,0x01
                              //最后一个字节随机，比如是 0x0c
```

所以我们在使用 transform_width 时需要注意输入的长度，如果有必要就手工补齐。

base64_from_binary 和 binary_from_base64

base64_from_binary 和 binary_from_base64 使用了 3.6.8 小节的转换迭代器 transform_iterator，调用一个函数对象实现了 base64 的编解码，它们位于头文件<boost/archive/iterators/base64_from_binary.hpp>和< boost/archive/iterators/binary_from_base64.hpp>，声明如下：

```
template<class Base>
class base64_from_binary :
```

```
    public transform_iterator<detail::from_6_bit<>, Base>
{...};
template<class Base>
class binary_from_base64 : public
    transform_iterator<detail::to_6_bit<>, Base>
{...};
```

base64 编码时需要先使用 transform_width<char*,6,8>转换位数，然后再用 base64_from_binary 编码，示范 base64 编码的代码如下：

```
char c1[4] = {1,2,3,4};                     //输入数据
vector<char> v1, v2;                        //两个 vector 容器

using namespace boost::archive::iterators;  //名字空间
typedef base64_from_binary<
    transform_width<char*, 6, 8 >> from_bin; //定义编码迭代器

std::copy(                                  //把二进制编码为 base64
    from_bin(c1), from_bin(c1 + 3),         //迭代器起始和结束位置
    back_inserter(v1));                     // v1 中会得到编码后的数据“AQID”
```

base64 解码时要以编码相反的顺序，先使用 binary_from_base64 解码，然后用 transform_width<*Iter*,8,6>转换为原来的形式：

```
typedef transform_width<                    //定义解码迭代器
    binary_from_base64<                     //迭代器使用 vector::iterator
        vector<char>::iterator>, 8, 6> from_b64;
std::copy(                                  //base64 解码为二进制
    from_b64(v1.begin()),from_b64(v1.end()), //迭代器起始和结束位置
    back_inserter(v2));                     //v2 中会得到解码后的数据

assert(v2.size() == 3 && v2.front() == 1);  //验证解码结果
```

其他功能

serialization 库里还有一个 insert_linebreaks 迭代器，它可以指定字符数插入换行符，在编码大量的数据时能够使格式更好看，示范代码如下：

```
std::copy(
     insert_linebreaks<from_bin, 4>(c1),     //每 4 个字符就换行
     insert_linebreaks<from_bin, 4>(c1 + 6),
     back_inserter(v1));
```

不过 serialization 库没有提供一个相反操作的 remove_linebreaks，所以在解码时我们需要使用标准算法 std::remove 移除字符串中的换行符，或者自行编写一个这样的迭代器。

另外，serialization 库的 base64 编码功能还有所欠缺，要求输入长度必须是 3 字节的整数倍，不能是任意长度，读者可自行尝试实现对它的填充和解填充，填补字符使用“=”。

9.9.5　base16 编解码

模仿 base64 的实现原理也可以实现十六进制的编解码，这个功能已经在第 3 章作为例子实现了多次，现在我们再来实践一下。

base16 编码

我们首先要实现一个函数对象 from_4_bit，它把一个 4 位的数字转换成 ASCII 码：

```
template<typename CharType>
struct from_4_bit {
  typedef CharType result_type;
  CharType operator()(CharType t) const{
      const char * lookup_table = "0123456789ABCDEF";    //16 进制转换表
      assert(t <= 15);
      return lookup_table[static_cast<size_t>(t)];       //查找码表转换
  }
};
```

接下来我们实现一个 basex_from_binary，它比 base64_from_binary 多了一个模板参数 FromBit，可以传入任意的函数对象，不必绑定成 base16 或者 base64：

```
template<typename Base, typename FromBit,
    typename CharType = boost::iterator_value<Base>::type>
class basex_from_binary :                                 //简化了一些代码
  public transform_iterator<FromBit,Base>
{
  friend class boost::iterator_core_access;
  typedef transform_iterator<FromBit,    Base> super_t;
public:
  template<class T>
  basex_from_binary(T start) :
      super_t(Base(start, FromBit()  )){}
};
```

现在我们就可以使用这两个类实现十六进制的编码：

```
unsigned char c1[4] = {0x12,0xab,0x97,0xef};      //字节数组

using namespace boost::archive::iterators;
typedef
    basex_from_binary<                            //定义编码迭代器
        transform_width<unsigned char*, 4, 8>,    //转换位数，注意迭代器
        from_4_bit<char>                          //转换函数对象
    >   trans_16;

std::copy(                                        //调用 copy 算法实现十六进制编码
    trans_16(c1), trans_16(c1 + 4),
    ostream_iterator<char>(cout));                //输出到屏幕
```

base16 解码

仿照 binary_from_base64 的方式，我们可以先实现一个从 ASCII 码到十六进制数的转换函数对象 to_4_bit，它把十六进制的 ASCII 码（'0'到'F'）转换成十六进制数：

```
template<class CharType>
struct to_4_bit {
  typedef CharType result_type;
  CharType operator()(CharType t) const
  {
      const char lookup_table[] = {                          //转换码表
          -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
          -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
          -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
           0, 1, 2, 3, 4, 5, 6, 7, 8, 9,-1,-1,-1,-1,-1,-1,   //0-9 的数字
          -1,10,11,12,13,14,15,-1,-1,-1,-1,-1,-1,-1,-1,-1,   //大写字母
          -1,10,11,12,13,14,15,-1,-1,-1,-1,-1,-1,-1,-1,-1,   //小写字母
          -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
          -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
      assert((unsigned)t <= 127);

      signed char value = -1;
      value = lookup_table[(unsigned)t];                     //查找码表转换
      assert(value != -1);

      return value;
```

```
    }
};
```

使用 to_4_bit 函数对象的迭代器类 binary_from_basex 实现如下：

```
template< typename Base, typename ToBit,
  typename CharType = boost::iterator_value<Base>::type>
class binary_from_basex :
  public transform_iterator<ToBit, Base>
{
  friend class boost::iterator_core_access;
  typedef transform_iterator<ToBit, Base> super_t;
public:
  template<class T>
  binary_from_basex(T  start) :
      super_t(Base(start, ToBit()){}
};
```

现在 base16 编解码的工作就全部完成了，下面的代码验证它们的工作效果：

```
... //之前已经完成了 base16 编码，输出到 vector<char> v1

typedef transform_width<  //定义 base16 解码迭代器，注意迭代器的使用
    binary_from_basex<char*, to_4_bit<char> >,    // base16 解码
    8, 4> untrans_16;                             //解码后再转换位数

vector<unsigned char> v2;                         //保存解码结果
std::copy(                  //调用 copy 算法，没有使用 vector 的迭代器，而是 char*指针
    untrans_16(&v1[0]), untrans_16(&v1[0] + v1.size()),
    back_inserter(v2));                           //输出到 v2

//验证解码结果
assert(v2.size() == 4);
assert(v2.back() == 0xef);
```

9.10 总结

本章中我们研究了 boost.serialization，它基于 C++98 标准和高效的代码构造了一个功能强大且完善的序列化库，弥补了 C++中不提供序列化功能的遗憾。虽然已经存在了很多其他可用的序列化库（如 Google 的 protobuf），但它绝对是其中的佼佼者。

serialization 库确立了存档、可序列化等概念，成功地分离了对象的序列化和存档格式的表示，并使用流处理实现了存档数据的输入输出，带来了最大的灵活性：<boost/archive/>目录下的存档的实现源码，而<boost/serialization/>目录下是各种数据类型的序列化实现源码。

存档的行为很类似流，可以使用相同风格的 operator<<或 operator>>来序列化或反序列化数据，这极大地简化了序列化的工作。serialization 库提供了三种存档格式：纯文本、XML 和二进制，这三种格式中最常用的是纯文本格式，它的可移植性最好，也可以跨平台交换序列化后的数据。纯文本格式的一个缺点是序列化后的数据量较大，但可以搭配流处理压缩功能来无缝地压缩大小。

serialization 库提供了对很多现有类型的序列化支持，包括基本类型，标准类型和标准容器，以及部分 Boost 类型和容器。虽然它还不能够对 Boost 中的所有容器和数据结构提供支持，但基于自身定义良好的扩展机制可以很容易地实现，只要为这些类型实现成员函数或者自由函数 serialize()就可以了，就此我们进行了详细的讨论。

指针的序列化是一个比较特别的议题，因为它涉及内存的动态创建，serialization 库为此做了许多工作以实现指针序列化的完美支持，它不仅能够序列化原始指针，也能够序列化智能指针和指针容器。

serialization 库的内容相当丰富，除了序列化还有许多其他的内容，本章的最后简单介绍了其中的部分有用的小工具，但远非全部，库里还有很多更深层次的功能等待读者去发掘，比如 pimpl 惯用法、自定义存档类等。

第10章

泛型编程

泛型编程是进入新世纪以来 C++的主流编程范式，它带来了更好更快的代码，但同时离早期的编程概念也越来越远，我们更多地是和类型打交道，代码编写工作更像是数学中的“代数”——真实的类型用占位符 T、U 等来代替，然后让编译器实例化模板去求解这些难题——这最终导致了模板元编程的诞生。

本章讨论 Boost 库中的四个泛型编程用的工具：

- enable_if ：它使用 SFINAE 原则，可以在编译期启用或禁用特定的泛型代码；
- call_traits ：它是一个非标准元函数，计算类型 T 可能的多种类型，经常被用于函数的入口参数或者返回值类型计算；
- concept_check ：它以库的方式实现了泛型编程中急需的概念检查功能，在 C++提供语言级别的概念检查支持之前是我们唯一可用的工具；
- function_types：它是一个类似 type_traits 的特征萃取库，但专门针对函数类型。

10.1 enable_if

enable_if 主要用来解决模板函数或模板类的重载解析问题，它允许模板函数或模板类仅针对某些特定类型有效，即依据条件启用或禁用某些特化形式。

enable_if 库位于名字空间 boost，为了使用 enable_if 组件，需要包含头文件 <boost/utility/enable_if.hpp>，即：

```
#include <boost/utility/enable_if.hpp>
using namespace boost;
```

10.1.1 类摘要

enable_if 库提供了两类共八个元函数，分别是 enable_if<>的“启用”系列和 disable_if<>的“禁用”系列。

enable_if<>的类摘要如下：

```
template <bool B, class T = void>              //T 缺省值是 void 类型
struct enable_if_c {
  typedef T type;                              //默认返回类型 T
};
template <class T>
struct enable_if_c<false, T> {};               //对 false 特化，无::type 返回

template <class Cond, class T = void>
struct enable_if :
    public enable_if_c<Cond::value, T> {};     //计算元函数 Cond
```

enable_if<>使用元函数转发技术，计算条件元函数 Cond 的值，再交给 enable_if_c<>。如果条件为 true，那么 enable_if<>/enable_if_c<>将返回类型 T，否则 enable_if<>/enable_if_c<>将是一个无返回的元函数。

disable_if<>与 enable_if<>相似，但在语义上是反义词，即条件 Cond 成立时无返回：

```
template <bool B, class T = void>              //T 缺省值是 void 类型
struct disable_if_c {
  typedef T type;                              //默认返回类型 T
};
template <class T>
struct disable_if_c<true, T> {};               //对 true 特化，无::type 返回

template <class Cond, class T = void>
struct disable_if :
    public disable_if_c<Cond::value, T> {};    //计算元函数 Cond
```

enable_if/disable_if 的工作原理涉及到 C++中模板实例化的重载解析。

处理重载函数时编译器要构造所有同名函数的集合，再从中选择一个最恰当的函数。当存在模板函数时，如果模板函数可以被模板实参推演实例化，那么它就是一个候选函数；反之，如果某个参数或返回值类型无效导致推演失败无法实例化，那么这个模板函数则不是候选函数，编译器也不会认为是一个编译错误。这就是著名的 SFINAE 原则，即“替代失败不是错误”(substitution failure is not an error)。

10.1.2 应用于模板函数

作为模板推演时的控制条件，enable_if 通常需要配合 type_traits 或者 mpl 使用来检查类型 T 是否满足某些条件，可以放在模板函数参数列表的最末尾用作缺省参数，或者是用作返回值，两种形式的效果是相同的，但有的时候只能使用一种形式，比如用于构造函数和析构函数时没有返回值，用于操作符重载时不能变动参数的数量。

下面的代码示范了 enable_if 的用法，这个 print() 函数仅在类型是整数时才生效：

```
template<typename T>
T print(T x ,
          typename enable_if<is_integral<T> >::type* =0) //整数时启用
{
  cout << "int:" << x << endl;
  return x;
}
```

代码中 enable_if 作为函数 print() 的缺省参数出现，声明了一个无名指针参数，默认值是空指针。这样，当编译器进行模板实例化时，如果 T 不是整数，那么 enable_if<> 将不会返回任何类型，导致实例化失败，从而使这个 print() 被禁用。

enable_if 的返回值用法如下，效果与缺省参数的形式相同：

```
template<typename T>
typename enable_if<is_integral<T>, T >::type //整数时启用
print(T x )
{ ...}
```

注意在这里我们向 enable_if<>传递了第二个元参数 T，因为如果不这么做的话 enable_if<>的返回值将是 void，不符合函数的签名，这与 enable_if<>的缺省参数用法略微有些不同[①]。第二个元参数也不一定必须是 T，我们也可以在这里再进行元计算，比如

① 当然 enable_if<>的缺省参数用法也可以使用 enable_if<is_integral<T>, T >的形式，但因为指针参数并不被实际使用，因此默认的 void 类型就可以正常工作了。

使用 promote<T>提升 T 的范围。

使用 disable_if<>可以禁止函数的实例化，例如不允许 print()操作类类型：

```
template<typename T>
typename disable_if<is_class<T>, T >::type          //T 是 class 时禁用
print(T x )
{ ...}
```

下面的这个例子摘自 ptr_container 库的 ptr_sequence_adapter 类(5.3 小节)，它使用了 disable_if<>，当类型是指针或者整数时禁用该函数：

```
template< class Range >
disable_if< is_pointer_or_integral<Range> >::type
insert( iterator before, const Range& r )
{
    insert( before, boost::begin(r), boost::end(r) );//使用 range 操作
}
```

多索引容器的键提取器（7.4 小节）也使用了 disable_if<>，它被用来递归地生成解引用指针类型的成员函数。

10.1.3 应用于模板类

enable_if 的启用或禁用模板类偏特化的用法与模板函数用法类似，它需要为类的模板参数列表增加一个额外缺省参数，缺省值是 void，然后再使用 enable_if 来偏特化。

示范 enable_if 的模板类用法的代码如下：

```
template<typename T, typename Enable = void>     //增加一个模板参数
class demo_class
{ ...};
template<typename T>                             //然后使用 enable_if 偏特化
class demo_class<T, typename enable_if<is_arithmetic<T> >::type>
{ ...};
```

在这里 demo_class 使用 enable_if 对 int、double 等算术类型进行了偏特化。很显然，enable_if 使得模板偏特化的应用范围更大了，可以针对某一些而不是某一个特定的类型偏特化，简单的偏特化相当于使用 is_same<>：

```
//对 string 类型特化
```

```
template<typename T>
class demo_class<T, typename enable_if<is_same<T, string> >::type>
{ ...};
```

10.1.4 lazy_enable_if

enable_if 库还提供另外四个功能类似的 lazy 版本，它们与同名的版本没有太多的不同，只是要求类型 T 必须有一个内部的::type 类型定义。

lazy_enable_if<>和 lazy_enable_if_c<>的定义如下：

```
template <bool B, class T>
struct lazy_enable_if_c {
  typedef typename T::type type;  //注意这里
};
template <class T>
struct lazy_enable_if_c<false, T> {};

template <class Cond, class T>
struct lazy_enable_if :
     public lazy_enable_if_c<Cond::value, T> {};
```

相对于 enable_if 来说，lazy_enable_if 对类型 T 增加了更强的约束，如果需要类型有内嵌的 type 类型，那么就使用它。

10.2 call_traits

call_traits 是一个很小的泛型工具，它封装了 C++中编写函数时可能是“最好的”传递参数给函数的方式，会自动推导出最高效的传递参数的类型，而且保证不会出现“引用的引用”[①] 这个非法的错误，某种程度上可以说是一个“智能参数类型”。

call_traits 位于名字空间 boost，为了使用 call_traits 组件，需要包含头文件 <boost/call_traits.hpp>，即：

```
#include <boost/call_traits.hpp>
using namespace boost;
```

① 在 C++98 中“引用的引用”（即 T&&）是非法的，但在 C++0x 中对此作出了修正，“引用的引用”仍然是一个引用，与 call_traits<>的解决方案相同。

10.2.1 类摘要

call_traits<>是一个返回多个值的非标准元函数，类摘要如下：

```
template <typename T>
struct call_traits
{
public:
   typedef T              value_type;             //T 的值类型
   typedef T&             reference;              //T 的引用类型
   typedef const T&       const_reference;        //T 的 const 引用类型
   typedef some_define    param_type;             //T 的被调用参数类型
};
```

这段代码只是 call_traits<>的基本形式，它还同时提供针对 T&、T[N]和 const T[N]的另外三个特化形式，代码与之类似。

10.2.2 用法

在实际的编程工作中经常需要编写函数相关参数，有很多规则告诉我们如何书写会使代码更加高效，比如 POD 类型通常传值比传引用或者指针更高效，而类类型通常应该传引用，为了更安全起见有时候还需要加上 const 修饰，函数返回时有值和引用的区别等，虽然规则并不多也不难理解，但实际使用的时候总是难免有所偏差，毕竟每个人对规则的理解程度都不一样，而且还存在偶尔笔误的可能。

使用 call_traits<>我们就可以不必过多地去考虑类型的各种形式，直接交给它来处理。call_traits<>对类型 T 执行元编程计算，依据 C++社区已经达成的共识返回以下四个最高效的类型（元数据）任我们选用：

- value_type ：T 的“值类型”，通常是 T，但对于数组 T[N]则退化为 const T*，可用于保存值或者以值返回；
- reference ：T 的“引用类型”，通常是 T&，可用于返回值引用；
- const_reference ：T 的“常引用类型”，通常是 const T &，可用于返回值引用；
- param_type ：T 的“参数类型”，通常是 const T&，但对于 POD 类型或者指针类型则是 const T，可用于作为“最好的”函数参数传递类型。

这四个类型中最方便也是最常使用的是 param_type，它可以用作函数的参数类型。

假设我们现在要编写一个将两个字符串相加的函数 scat()：

```
string scat(string& s1, string& s2)
{ return s1 + s2;}
```

初看上去函数并无多大缺陷，参数的传递使用了引用，避免了大类的拷贝代价，返回也使用了值返回，不会出现悬空引用。但这个函数不是最佳的，接口存在一点小缺陷，下面的简单代码无法通过编译：

```
cout << scat("1", "2");                              //编译错误
```

这是因为编译器无法把一个字符串类型转换为 string&类型，我们必须使用临时变量的形式来写，显得麻烦许多：

```
cout << scat(string("1"), string("2"));
```

使用 call_traits<>我们无须费力就可以避免这样的“低级错误”，使我们的函数接口更加合理且高效：

```
call_traits<string>::value_type                      //返回值类型
scat(call_traits<string>::param_type s1,             //参数类型
   call_traits<string>::param_type s2)               //参数类型
{
  //断言参数类型是引用，并且是 const string&
  assert(is_reference<call_traits<string>::param_type>::value);
  assert((is_same<call_traits<string>::param_type,
                 const string&>::value));

  return s1 + s2;
}
```

对于模板类来说 call_traits<>也非常有用，它可以很好地推断最合适的模板参数类型，极大地节约我们定义类型的时间和精力。例如，下面的代码定义了一个模板类 demo_class，得益于 call_traits<>的使用，它可以容纳任意的类型，包括数组和引用：

```
template<typename T>
class demo_class
{
public:
```

```
    typedef typename call_traits<T>::value_type v_type;
    typedef typename call_traits<T>::param_type p_type;
    typedef typename call_traits<T>::reference  r_type;
    typedef typename call_traits<T>::const_reference cr_type;
  private:
    v_type v;
  public:
    demo_class(p_type p):
      v(p){}
    v_type value()
    {   return v;   }
    r_type get()
    {   return v;   }
};
```

示范 demo_class 的用法的代码如下：

```
int a[3] = {1,2,3};
demo_class<int[3]> di(a); //容纳数组类型
assert(di.value()[0] == 1);

char c = 'A';
demo_class<char&> dc(c);  //容纳引用类型
assert(dc.get() == c);
```

读者可以自行尝试一下，如果不使用 call_traits<>，编写这样一个可以容纳任意类型的模板类是一个多么复杂的工作。

10.2.3 实现原理

call_traits<>的实现原理并不复杂，它使用模板特化技术，针对 T&、T[N]和 const T[N]三种类型做了特殊处理，解决了“引用的引用”和数组的类型问题。例如，对于 T&，call_traits<>的特化代码如下：

```
template <typename T>
struct call_traits<T&>         //针对 T&特化
{
    typedef T& value_type;
    typedef T& reference;      //注意这里
    typedef const T& const_reference;
```

```
    typedef T& param_type;
};
```

通过元编程 call_traits<>这个间接层将 T&的引用类型仍然定义为 T&，因而成功地回避了“引用的引用”错误。

对于一般的情况，call_traits<>使用位于名字空间 boost::detail 里的两个元函数 ct_imp<>和 ct_imp2<>来实现对 param_type 的类型计算。ct_imp<>的定义如下：

```
template <typename T, bool isp, bool b1>
struct ct_imp
{
    typedef const T& param_type;
};
```

ct_imp<>有三个元参数，T 是要计算的类型，isp 和 b1 是两个 bool 值，分别表示 T 是否为指针、是否为算术类型，对应 type_traits 库的元函数则是 is_pointer<T>和 is_arithmetic<T>。如果 T 既不是指针也不是算术类型（元参数均为 false），那么 param_type 就被定义为常引用类型 const T&，否则 ct_imp<>进行模板特化：

```
template <typename T, bool b1>
struct ct_imp<T, true, b1>        //指针类型
{
    typedef const T param_type;
};
template <typename T, bool isp>
struct ct_imp<T, isp, true>       //算术类型
{
    typedef typename ct_imp2<T,    //元函数转发
        sizeof(T) <= sizeof(void*)>::param_type param_type;
};
```

元函数 ct_imp2<>中再根据类型 T 是否是一个“小”类型（小于一个指针的大小）来决定 param_type 是 const T 或是 const T&：

```
template <typename T, bool small_>
struct ct_imp2
{
    typedef const T& param_type;
};
```

```
template <typename T>
struct ct_imp2<T, true>
{
   typedef const T param_type;
};
```

10.3 concept_check

泛型编程中使用的是“静态多态”，它在语义上经常要求类型具有某种“特征”或者满足某种“条件”，例如有内嵌的类型定义或者固定名字的成员函数、支持迭代操作，这些要求通常被称为“概念”。

但 C++最初并不是为泛型编程所设计的，它对运行期检查做的很好，但对泛型编程缺乏足够的支持，没有有效的对模板类型参数的验证机制和手段，这给编写泛型程序带来了很大的不方便。C++11 曾经有提案要求在语言级别增加对“概念检查”的支持，但非常遗憾在最后关头被否决了，不过 Boost 的 concept_check 以库的方式达到了同样的效果，并且能够在编译出错时给出更可读的信息。

concept_check 位于名字空间 boost，为了使用 concept_check 组件，需要包含头文件<boost/concept_check.hpp>，即：

```
#include <boost/concept_check.hpp>
using namespace boost;
```

10.3.1 概述

概念检查的基本工具是宏 BOOST_CONCEPT_ASSERT，它的用法很像静态断言 BOOST_STATIC_ASSERT，可以用在任何域（scope）：函数域、类域、名字空间域。如果概念检查不通过则导致编译错误，但 BOOST_CONCEPT_ASSERT 与静态断言也是有区别的，不能在宏中使用逻辑运算符（!、&&等）。还需要注意的是宏的参数必须要用括号括起来，也就是说使用双重括号，像这样：

```
BOOST_CONCEPT_ASSERT((some_check_class));    //双重括号
```

concept_check 库提供了大量的概念检查类用来检查类型是否符合某个概念，这与 type_traits 库（1.2 小节）有些类似，它们同样是检测类型的属性，只是 type_traits 的检查更偏重于 C++的类型系统，而 concept_check 库更偏重于类型的功能属性。另外，type_traits 库提供的是标准的元函数，而 concept_check 库的概念检查类虽然也可以算做是元函数，它们可以但通常不用于元计算，多数情况用来配合检查宏工作。

concept_check 库里用于概念检查元函数有很多，可分为如下五个类别：

- 基本的概念检查　　：检查整数类型、拷贝构造、缺省构造、赋值函数等基本的概念；
- 函数对象概念检查　：检查函数对象相关的概念；
- 标准迭代器概念检查：检查标准库的五种迭代器分类概念；
- 新式迭代器概念检查：检查 Boost 定义的九种迭代器分类概念（参见 3.1.3 小节）；
- 容器概念检查　　　：检查容器相关的概念。

10.3.2 基本概念检查

基本概念检查的功能与 type_traits 提供的功能很相似，包括如下的检查类：

- Integer<T>：检查 T 是否是内建的整数类型，相当于 is_integral<T>；
- SignedInteger<T>：检查 T 是否是内建的有符号整数类型，相当于 is_signed<T>；
- UnsignedInteger<T>：检查 T 是否是内建的无符号整数类型，相当于 is_unsigned<T>；
- Convertible<X, Y>：检查 X 是否可转换为 Y，相当于 is_convertible<X, Y>；
- Assignable<T>：检查 T 是否是可赋值的，要求有赋值操作符 operator=；
- SGIAssignable<T>：检查 T 是否符合 SGI 赋值概念，要求有拷贝构造函数和 operator=；
- DefaultConstructible<T>：检查 T 是否有缺省构造函数；
- CopyConstructible<T>：检查 T 是否有拷贝构造函数；
- EqualityComparable<T>：检查 T 是否可以进行相等比较，即定义了 operator==；
- LessThanComparable<T>：检查 T 是否可以进行小于比较，即定义了 operator<；
- Comparable<T>：检查 T 是否是可以进行所有关系运算，即定义了<、>、<=和>=。

我们可以使用这些概念检查类来实现标准库的 min() 函数：

```
template<typename T>
T my_min(const T& l,const T& r)
{
  BOOST_CONCEPT_ASSERT((LessThanComparable<T>));     //要求可小于比较
  BOOST_CONCEPT_ASSERT((SGIAssignable<T>));          //要求可拷贝赋值

  return (l < r) ? l : r;
}
```

这样，当一个没有定义 operator<的类被传入 my_min() 时会发生编译错误：

```
complex<double> cp1, cp2;
my_min(cp1, cp2);                                    //编译错误
```

错误信息可能是这样：

```
concept_check.hpp : binary '<' : 'stlp_std::complex<double>' does not define
this operator or a conversion to a type acceptable to the predefined operator
```

这段错误信息用明确的出错位置（concept_check.hpp）和信息表明了代码违反了概念检查，有助于我们迅速定位问题的原因所在。

10.3.3 函数对象概念检查

函数对象概念检查主要基于标准库的函数对象定义，它们的模板参数较复杂，除了输入要检查的函数对象类型外，还依情况需要输入返回值类型和参数类型。下面的列表中 F 是函数对象类型，R 是返回值类型，A 和 B 分别是以下两个参数类型：

- Generator<F, R> ：检查 F 是否是无参函数对象；
- UnaryFunction<F,R,A> ：检查 F 是否是单参函数对象；
- BinaryFunction<F,R,A,B> ：检查 F 是否是双参函数对象；
- UnaryPredicate<F,A> ：检查 F 是否是单参谓词（返回 bool 类型）；
- BinaryPredicate<F,A,B> ：检查 F 是否是双参谓词；
- Const_BinaryPredicate<F,A,B> ：检查 F 是否是 const 双参谓词；

- AdaptableGenerator<F,R> ：检查 F 是否是且有内嵌 result_type 定义的无参函数对象，因而可以被 std::bind1st 等函数对象适配器使用；
- AdaptableUnaryFunction<F,R,A> ：检查 F 是否是有内嵌 result_type 定义的单参函数对象；
- AdaptableBinaryFunction<F,R,A,B>：检查 F 是否是有内嵌 result_type 定义的双参函数对象；
- AdaptablePredicate<F,A> ：检查 F 是否是有内嵌 result_type 定义的单参谓词；
- AdaptableBinaryPredicate<F,A,B> ：检查 F 是否是有内嵌 result_type 定义的双参谓词。

示范这些概念检查类的代码如下：

```
BOOST_CONCEPT_ASSERT((UnaryFunction< negate<int>, int, int>));
BOOST_CONCEPT_ASSERT((AdaptableUnaryFunction< negate<int>, int, int>));
BOOST_CONCEPT_ASSERT((BinaryFunction< plus<int>,int, int, int>));
```

10.3.4 标准迭代器概念检查

concept_check 的迭代器概念检查完全依据 C++98 标准的定义（参见 3.1.2 小节），概念检查类还定义了 value_type、reference、pointer 等内部类型，等价于 std::iterator_traits<>。

迭代器概念检查类列表如下：

- InputIterator<I> ：检查 I 是否是输入迭代器；
- OutputIterator<I,T> ：检查 I 是否是输出类型 T 的输出迭代器；
- ForwardIterator<I> ：检查 I 是否是前向迭代器；
- Mutable_ForwardIterator<I> ：检查 I 是否是可变前向迭代器（即可修改，支持*i++ = *i 操作）；
- BidirectionalIterator<I> ：检查 I 是否是双向迭代器；
- Mutable_BidirectionalIterator<I>：检查 I 是否是可变双向迭代器；

- RandomAccessIterator<I> ：检查 I 是否是随机访问迭代器；
- Mutable_RandomAccessIterator<I> ：检查 I 是否是可变随机访问迭代器。

示范这些概念检查类用法的代码如下：

```
//原生指针满足迭代器的概念
BOOST_CONCEPT_ASSERT((InputIterator<int*>));
BOOST_CONCEPT_ASSERT((OutputIterator<int*, int>));
BOOST_CONCEPT_ASSERT((RandomAccessIterator<int*>));

//可以从概念检查类获取迭代器的类型定义
assert((is_same<InputIterator<int*>::pointer,
  iterator_traits<int*>::pointer>::value));

//slist 是 STLport 提供的单向链表容器，支持前向迭代
BOOST_CONCEPT_ASSERT((ForwardIterator<slist<int>::iterator>));
BOOST_CONCEPT_ASSERT((Mutable_ForwardIterator<slist<int>::iterator>));

//vector 支持双向迭代器和随机访问
typedef vector<int>::iterator I;
BOOST_CONCEPT_ASSERT((BidirectionalIterator<I>));
BOOST_CONCEPT_ASSERT((RandomAccessIterator<I>));
```

同样的，我们可以把它应用于自己的代码，例如下面的_sort()函数包装了标准库的stable_sort()，增加了概念检查，要求必须是可随机访问的迭代器：

```
template <typename I>
void _sort(I first, I last)
{
  BOOST_CONCEPT_ASSERT((RandomAccessIterator<I>));
  std::stable_sort(first, last);
}
```

10.3.5 新式迭代器概念检查

iterators 库定义了一组新式的迭代器概念（参见 3.1.3 小节），同时也提供了相应的概念检查类，也同样可以用作 traits 类来使用。

这些概念检查类位于名字空间 boost_concepts（注意，不是 boost），需要包含头文件<boost/iterator/iterator_concepts.hpp>，即：

```
#include <boost/iterator/iterator_concepts.hpp>
```

新式迭代器概念检查类如下：

- ReadableIteratorConcept<I> ：检查 I 是否是可读迭代器；
- WritableIteratorConcept<I,T> ：检查 I 是否是可写迭代器；
- SwappableIteratorConcept<I> ：检查 I 是否是可交换迭代器；
- LvalueIteratorConcept<I> ：检查 I 是否是左值迭代器；
- IncrementableIteratorConcept<I> ：检查 I 是否是可递增迭代器；
- SinglePassIteratorConcept<I> ：检查 I 是否是单遍迭代器；
- ForwardTraversalConcept<I> ：检查 I 是否是前向迭代器；
- BidirectionalTraversalConcept<I>：检查 I 是否是双向迭代器；
- RandomAccessTraversalConcept<I> ：检查 I 是否是随机访问遍历迭代器。

下面的代码使用这些新式迭代器概念检查类检查 vector<bool>::iterator 和 3.4.3 小节实现的 vs_iterator：

```
using namespace boost_concepts;        //打开名字空间
typedef vector<bool>::iterator I;

// vector<bool>::iterator 是可交换的随机访问迭代器
BOOST_CONCEPT_ASSERT((ReadableIterator<I>));
BOOST_CONCEPT_ASSERT((WritableIterator<I>));
BOOST_CONCEPT_ASSERT((SwappableIteratorConcept<I>));
BOOST_CONCEPT_ASSERT((RandomAccessTraversalConcept<I>));

//检查 vs_iterator，可读可写可交换的单遍迭代器
BOOST_CONCEPT_ASSERT((ReadableIterator<vs_iterator<int> >));
BOOST_CONCEPT_ASSERT((WritableIterator<vs_iterator<int> >));
BOOST_CONCEPT_ASSERT((SwappableIterator<vs_iterator<int>>));BOOST_CONCEPT_
ASSERT((SinglePassIterator<vs_iterator<int> >));

//不是左值迭代器，编译错误
BOOST_CONCEPT_ASSERT((LvalueIteratorConcept<I>));
```

因为新式迭代器概念兼容标准库迭代器概念，因此标准迭代器概念检查类相当于是新式迭代器概念检查类的组合，读者可参考 10.3.8 小节。

10.3.6 容器概念检查

容器概念检查类检查是否符合标准库的容器定义，也就是说是否具有 being()、end()、empty()、size()等成员函数，是否有若干容器必备的内嵌类型定义。

这些容器概念检查类都有内部的 value_type、reference 等概念满足的类型定义，因此也可以把它们当做是容器的 traits 元函数。

基本的容器概念检查类如下：

- Container<C> ：检查 C 是否满足标准容器定义；
- Mutable_Container<C> ：检查 C 是否满足可变容器定义（即可以修改元素的值）；
- ForwardContainer<C> ：检查 C 是否可以前向迭代；
- Mutable_ForwardContainer<C> ：检查 C 是否满足可变前向迭代容器定义；
- ReversibleContainer<C> ：检查 C 是否可以逆向迭代；
- Mutable_ReversibleContainer<C> ：检查 C 是否满足可变逆向迭代容器定义；
- RandomAccessContainer<C> ：检查 C 是否满足随机访问容器定义；
- Mutable_RandomAccessContainer<C>：检查 C 是否满足可变随机访问容器定义。

下面的代码检查了一些标准容器和 Boost 容器：

```
BOOST_CONCEPT_ASSERT((Container<vector<int>>));
BOOST_CONCEPT_ASSERT((RandomAccessContainer<vector<int>>));
BOOST_CONCEPT_ASSERT((Container<slist<int>>));

//容器检查类有相应的类型定义
assert((is_same<vector<int>::value_type,
            Container<vector<int>>::value_type>::value));

BOOST_CONCEPT_ASSERT((ForwardContainer<slist<int>>));
//slist 是单向链表，不能逆向迭代，编译错误
```

```
BOOST_CONCEPT_ASSERT((ReversibleContainer<slist<int>>));

//循环缓冲容器 circular_buffer 符合标准容器定义
BOOST_CONCEPT_ASSERT((Container<boost::circular_buffer<int> >));
//array 虽然很像容器，但它不符合标准容器定义，编译错误
BOOST_CONCEPT_ASSERT((Container<boost::array<int> >));
```

接下来的概念检查类检查容器的序列类型：

- Sequence<C> ：检查 C 是否是线性序列容器，如 vector、deque；
- FrontInsertionSequence<C> ：检查 C 是否支持序列头插入操作；
- BackInsertionSequence<C> ：检查 C 是否支持序列尾插入操作；
- AssociativeContainer<C> ：检查 C 是否是关联容器；
- UniqueAssociativeContainer<C> ：检查 C 是否不允许重复键；
- MultipleAssociativeContainer<C>：检查 C 是否允许重复键；
- SimpleAssociativeContainer<C> ：检查 C 是否键即值，即集合类型；
- PairAssociativeContainer<C> ：检查 C 是否是键-值关联类型，即映射；
- SortedAssociativeContainer<C> ：检查 C 是否是有序的。

这些概念检查类使用方法如下：

```
BOOST_CONCEPT_ASSERT((Sequence<vector<int>>));
BOOST_CONCEPT_ASSERT((Sequence<deque<int>>));
BOOST_CONCEPT_ASSERT((Sequence<list<int>>));

BOOST_CONCEPT_ASSERT((FrontInsertionSequence<deque<int>>));
BOOST_CONCEPT_ASSERT((BackInsertionSequence<list<int>>));

BOOST_CONCEPT_ASSERT((AssociativeContainer<set<int>>));
BOOST_CONCEPT_ASSERT((AssociativeContainer<map<int,int>>));
BOOST_CONCEPT_ASSERT((MultipleAssociativeContainer<multimap<int,int>>));

BOOST_CONCEPT_ASSERT((SimpleAssociativeContainer<set<int>>));
BOOST_CONCEPT_ASSERT((SortedAssociativeContainer<set<int>>));
```

10.3.7 在函数声明中的概念检查

BOOST_CONCEPT_ASSERT 在基本的概念检查中工作的足够好，但仍然有不足，有时候我们希望能够在函数的声明中“显式”给出概念检查，让用户在使用接口时明确函数所需要的概念，这时 BOOST_CONCEPT_ASSERT 就无能为力了。

因此，concept_check 库另外提供一个宏 BOOST_CONCEPT_REQUIRES，它可以在模板函数的声明里做概念检查，把检查的时机更向前提一步。

要使用 BOOST_CONCEPT_REQUIRES 必须另外包含如下的头文件：

```
#include <boost/concept/requires.hpp>
```

宏 BOOST_CONCEPT_REQUIRES 的基本形式如下：

```
template <...>
BOOST_CONCEPT_REQUIRES(
  ((some_check_class 1))  //概念检查类列表开始
  ((some_check_class 2))
  ...
  ((some_check_class N)), //概念检查类列表结束
  (return type)        )  //返回值类型
function_name(...)        //函数名
{...}
```

BOOST_CONCEPT_REQUIRES 只能用在函数声明里，它有两个参数，第一个是概念检查类列表，是多个双重括号的序列，第二个是函数的返回类型，也必须用括号括起来。

使用 BOOST_CONCEPT_REQUIRES 可以把 10.3.2 小节的 my_min()改写如下：

```
template<typename T>
BOOST_CONCEPT_REQUIRES(
((LessThanComparable<T>))           //注意，检查列表没有逗号分隔
((SGIAssignable<T>)),               //检查结束使用逗号
(T)                       )         //返回类型是第二个宏参数
my_min(const T& l,const T& r)
{
  return (l < r) ? l : r;
}
```

10.3.4 小节对 stable_sort()的包装类_sort()也可以改用 BOOST_CONCEPT_REQUIRES，代码如下：

```
template <typename I>
BOOST_CONCEPT_REQUIRES(
((Mutable_RandomAccessIterator<I>))
((LessThanComparable<typename RandomAccessIterator<I>::value_type>))
,
(void)                        )
_sort(I first, I last)
{
  std::stable_sort(first, last);
}
```

10.3.8 概念原型类

为了方便测试验证泛型代码，concept_check 库提供了一些精确匹配标准库概念的最小类型——称为原型类（archetype class），因为它们足够小，因而能够比普通类（通常具备更多的特性）更严格地测试泛型代码。

使用概念原型需要包含额外的头文件①：

```
#include <boost/concept_archetype.hpp>
```

concept_check 库目前有基本概念原型、函数对象概念原型和迭代器概念原型，暂时还没有容器概念原型。原型类与概念检查类基本是一一对应的，故在此不做列举。

使用概念原型相当于用这些原型类自行构造出一套类型系统，然后把这个原型类型系统送入到泛型代码中进行测试。例如，下面的代码测试了 my_min()和_sort()函数：

```
//定义可拷贝、赋值和比较的基本原型
typedef null_archetype<> T;
typedef sgi_assignable_archetype<T> at;
typedef less_than_comparable_archetype<at> vt;

//使用 boost::detail::dummy_constructor 初始化
boost::detail::dummy_constructor dummy_cons;
```

① 对于新式迭代器概念原型需要包含头文件<boost/iterator/iterator_archetypes.hpp>，其用法与标准迭代器原型略有不同，需要制定值类型、访问类型和遍历类型三个模板参数。

```
vt v1(dummy_cons), v2(dummy_cons);

my_min(v1, v2);

//定义可变随机访问迭代器原型
typedef mutable_random_access_iterator_archetype<vt> rt;
rt begin, end;
_sort(begin, end);
```

这段代码中如果把 mutable_random_access_iterator_archetype 改为 random_access_iterator_archetype 那么将导致编译错误，因为原型类精确地描述了所需的概念。

下面的代码示范了标准库迭代器原型和 Boost 迭代器原型的测试（需要包含头文件 <boost/iterator/iterator_archetypes.hpp>）：

```
using namespace boost_concepts;
typedef sgi_assignable_archetype<> T;
//标准的输入迭代器
typedef input_iterator_archetype<T > I;

BOOST_CONCEPT_ASSERT((ReadableIterator<I>));
BOOST_CONCEPT_ASSERT((SinglePassIterator<I>));
BOOST_CONCEPT_ASSERT((InputIterator<I>));

//使用 Boost 的迭代器原型定义输入迭代器
typedef boost::iterator_archetype<T,
  boost::iterator_archetypes::readable_iterator_t,
  boost::single_pass_traversal_tag > II;

BOOST_CONCEPT_ASSERT((ReadableIterator<II>));
BOOST_CONCEPT_ASSERT((SinglePassIterator<II>));
BOOST_CONCEPT_ASSERT((InputIterator<II>));
```

10.4 function_types

function_types 库是一个 traits 工具，它专门处理函数元数据（函数类型，包括普通函数、函数指针、函数引用和成员函数指针等），较 type_traits::function_traits 处理的更加精细，不仅可以对函数元数据更精准的分类，还能够进行神奇的分解和合成——

或许它名字叫作 function_type_traits 更加恰当。

function_types 与 boost.result_of 不同：result_of 要求模板参数必须是一个完整的（含参数的）调用式，即形如 Func(T0, T1,...)的形式，而 function_types 则不要求函数调用式，只要求有一个可调用的函数类型，即 Func。另外，result_of 处理的目标要比 function_types 多，不仅包括内建的函数类型，还包括函数对象。

function_types 位于名字空间 boost::function_types，但它没有一个统一的包含头文件，因此在使用某个功能时必须包含特定的头文件。因为 function_types 中存在与 type_traits 的同名元函数，故接下来的代码必要时会加上名字空间限定，假定有：

```
namespace ft = boost::function_types;
using namespace ft;
```

10.4.1 属性标签

与 type_traits 直接用元函数 is_const<>、is_volatile<>等检测类型附加属性的做法不同，function_types 库使用一个特殊的类 property_tag 来标记，这样带来的好处是减小了接口函数的数量，而且可以更加容易地检查组合属性。

使用这些属性标签需要包含如下的头文件：

```
#include <boost/function_types/property_tags.hpp>
```

function_types 提供的基本属性标签列表如下：

- variadic/non_variadic ：函数类型有/无可变参数列表（即使用“...”）；
- const_qualified/non_const ：函数类型有/无 const 修饰；
- volatile_qualified/non_volatile ：函数类型有/无 volatile 修饰；
- const_non_volatile ：函数类型有 const 无 volatile 修饰；
- volatile_non_const ：函数类型有 volatile 无 const 修饰；
- cv_qualfied ：函数类型同时有 const 和 volatile 修饰；
- non_cv ：函数类型即没有 const 也没有 volatile 修饰；
- default_cc ：函数类型使用缺省调用约定。

这些属性标签通常用于配合 function_types 库的其他组件使用，例如下面的代码定义了三个函数类型，并用 ft::is_function<>元函数（参见 10.4.2 小节）检查：

```
typedef int  (Func0)(...);         //有可变参数列表，返回 int
typedef void(Func1)(int);          //参数是 int，无返回
typedef int  (Func2)() volatile;   //volatile 函数，无参数，返回 int

assert((ft::is_function<Func0, variadic>::value));
assert((ft::is_function<Func0, default_cc>::value));
assert((ft::is_function<Func1, non_cv>::value));
assert((!ft::is_function<Func2, const_qualified>::value));
assert((!ft::is_function<Func2, volatile_qualified>::value));
```

除了这 11 个基本属性标签，function_types 库还有两个特别的属性标签：null_tag 和 tag。

null_tag 是空对象模式的应用，它表示没有标签，被用作许多元函数的缺省模板参数，通常我们不会用到它。tag 则是一个元函数，用于组合基本属性标签构成复合属性，它最多能够接受四个参数，如果参数中的属性有冲突则使用最右边的一个。

示范 tag 用法的代码如下：

```
//组合 variadic 和 null_tag，相当于单独的 variadic
assert((ft::is_function<Func0, tag<variadic, null_tag> >::value));
//组合 variadic 和 non_cv
assert((ft::is_function<Func0, tag<variadic, non_cv> >::value));
//组合 non_cv 和 default_cc
assert((ft::is_function<Func0, tag<non_cv, default_cc> >::value));
//const_qualified 与 non_cv 冲突，以右边的 non_cv 为准
assert((ft::is_function<Func1, tag<const_qualified, non_cv> >::value));
```

10.4.2 函数类型分类

function_types 的函数类型分类功能包括八个元函数，用于处理函数类型、函数指针类型、成员函数指针类型等各种情况，需要包含如下的八个头文件：

```
#include <boost/function_types/is_function.hpp>
#include <boost/function_types/is_function_pointer.hpp>
#include <boost/function_types/is_function_reference.hpp>
#include <boost/function_types/is_member_function_pointer.hpp>
```

```
#include <boost/function_types/is_member_object_pointer.hpp>
#include <boost/function_types/is_member_pointer.hpp>
#include <boost/function_types/is_callable_builtin.hpp>
#include <boost/function_types/is_nonmember_callable_builtin.hpp>
```

这些分类元函数的声明如下，可以使用第二个模板参数指定检查的标签：

```
template<typename T, typename Tag = null_tag>
struct is_function;                         //检查是否是函数类型
template<typename T, typename Tag = null_tag>
struct is_function_pointer;                 //检查是否是函数指针类型
template<typename T, typename Tag = null_tag>
struct is_function_reference;               //检查是否是函数引用类型
template<typename T, typename Tag = null_tag>
struct is_member_function_pointer;          //检查是否是成员函数指针类型
template<typename T>
struct is_member_object_pointer;            //检查是否是成员变量指针类型
template<typename T, typename Tag = null_tag>
struct is_member_pointer;                   //检查是否是成员指针类型（变量或函数）
template<typename T, typename Tag = null_tag>
struct is_callable_builtin;                 //检查是否是可调用内建类型
template<typename T, typename Tag = null_tag>
struct is_nonmember_callable_builtin;       //检查是否是非成员可调用内建类型
```

ft::is_function<>等元函数的名称恰当地表明了它们的功能，用法与 type_traits 的 is_function<>很类似，例如：

```
typedef Func0& RFunc0;                      //函数引用类型
typedef Func1* PFunc1;                      //函数指针类型
typedef bool (string::*Func3)()const;       //成员函数指针类型

assert((ft::is_function_reference<RFunc0, variadic>::value));
assert((ft::is_function_pointer<PFunc1, non_cv>::value));
assert((ft::is_member_pointer<Func3, const_qualified>::value));
assert((ft::is_member_function_pointer<Func3>::value));
assert((ft::is_callable_builtin<PFunc1>::value));
assert((ft::is_nonmember_callable_builtin<RFunc0>::value));
```

10.4.3 函数类型分解

function_types 的函数类型分解功能与 type_traits::function_traits<>类

似，可以取出函数的返回类型、参数类型和参数数量等函数要素，它要求函数类型必须是可调用的内建类型，即 ft::is_callable_builtin<F>::value == true。

function_types 的函数类型分解功能需要包含如下的头文件：

```
#include <boost/function_types/result_type.hpp>
#include <boost/function_types/function_arity.hpp>
#include <boost/function_types/parameter_types.hpp>
#include <boost/function_types/components.hpp>
```

这些类型分解元函数如下：

- result_type<F>　　：以::type 返回函数 F 的返回值类型；
- function_arity<F>：以::value 返回函数 F 的参数数量，如果 F 是一个成员指针，那么隐藏的 this 参数也包括在内；
- parameter_types<F>：函数 F 的参数类型，是一个 mpl 类型容器，如果 F 是一个成员指针，那么隐藏的 this 参数也包括在内；
- components<F>　　：函数 F 的所有属性，包括返回值类型和参数类型，是一个 mpl 类型容器。

示范这四个元函数用法的代码如下，其中使用了部分 mpl 操作容器的元函数（参见 11.4 节）。

```
#include <boost/mpl/front.hpp>    //mpl 类型容器操作头文件
#include <boost/mpl/back.hpp>
#include <boost/mpl/size.hpp>

int main()
{
  //获得返回类型和参数数量
  assert((is_same<result_type<Func1>::type, void>::value)) ;
  assert(function_arity<Func1>::value == 1);

  //获得参数类型，存入一个 mpl 类型容器
  typedef ft::parameter_types<Func1> Fp1;
  //使用 mpl 元函数获得容器大小
  assert((mpl::size<Fp1>::value == 1));
  //使用 mpl 元函数 front 获得类型容器里的第一个元数据，即第一个参数类型
```

```
    assert((is_same<typename mpl::front<Fp1>::type, int>::value));

    //定义一个新的成员函数指针类型
    typedef bool (string::*Func4)(const char *, size_t);
    typedef ft::parameter_types<Func4> Fp4;
    assert((mpl::size<Fp4>::value == 3));
    //使用 mpl 元函数 front 获得类型容器里的第一个元数据，即 this 指针类型①
    assert((is_same<typename mpl::front<Fp4>::type, string&>::value));
    //使用 mpl 元函数 back 获得类型容器里的最后一个元数据，即最后一个参数类型
    assert((is_same<typename mpl::back<Fp4>::type, size_t>::value));

    //获得函数类型 Func1 的所有组成属性，存入 mpl 类型容器
    typedef ft::components<Func1>  C1;
    assert((mpl::size<C1>::value == 2));
    assert((is_same<typename mpl::front<C1>::type,
            result_type<Func1>::type>::value));
}
```

10.4.4 函数类型合成

函数类型合成是函数类型分解的逆操作，它可以把一个存储了若干类型的 mpl 容器（参见 11.4 节）重新组合成一个可调用的函数或者函数指针类型。

function_types 的函数类型合成功能需要包含如下的头文件：

```
#include <boost/function_types/function_type.hpp>
#include <boost/function_types/function_pointer.hpp>
#include <boost/function_types/function_reference.hpp>
#include <boost/function_types/member_function_pointer.hpp>
```

function_types 提供了四个用于合成函数类型的元函数，除了::type 返回的类型不同外它们的声明基本相同，例如，返回函数指针类型的 function_pointer<>声明如下：

```
template<typename Types, typename Tag = null_tag>
struct function_pointer;
```

function_pointer<>的第一个参数是 mpl 类型容器，它被用于合成函数指针类型，第二个参数是属性标签，用来附加所需的额外属性。

① 你也许会吃惊地发现，this“指针”原来并不是一个指针，而是一个引用。

示范函数类型合成的代码如下：

```
#include <boost/mpl/vector.hpp>

int main()
{
  //定义一个容纳 void、int 的容器，相当于函数 void(int)
  typedef mpl::vector<void, int> types1;

  //合成函数类型
  typedef ft::function_type<types1>::type Ft1;
  assert((is_same<Func1, Ft1>::value));

  //合成函数指针类型
  typedef ft::function_pointer<types1>::type Ftp1;
  assert((is_same<PFunc1, Ftp1>::value));

  //合成成员函数指针类型，有 const 修饰
  typedef ft::member_function_pointer<
      typename mpl::vector<bool, string&, void>::type,
      const_qualified
      >::type Ftp3;
  assert((is_same<Func3, Ftp3>::value));
}
```

10.4.5 其他议题

function_types 库默认支持的参数数量最大为 20 个，如果想变更这个数量，则可以在包含头文件前定义宏 BOOST_FT_MAX_ARITY，例如：

```
#define BOOST_FT_MAX_ARITY    5
...//function_types 的其他头文件
```

这将使 function_types 最多只能使用五个参数，可以略微减少一些预处理元编程所需要的时间。

10.5 总结

本章我们讨论了 Boost 库中的四个泛型编程组件：enable_if、call_traits、

concept_check 和 function_types，使用它们可以很好地改善泛型代码的质量。

enable_if 用于编写模板函数或模板类，在编写泛型代码时可以主动地控制编译器的行为，把不希望出现的形式从重载决议中去掉。它比简单地使用静态断言效果更好，因为静态断言仅仅是验证了某些编译期条件，不能阻止函数或模板类的重载形式。Boost 库的许多组件都使用了 enable_if，虽然我们在实际开发中不一定会用到它，但理解它可帮助我们深入理解其他 Boost 组件的工作原理。

call_traits 是一个比较简单的泛型编程工具，它可以计算类型 T 相关的各种类型，所以很有用，特别是在大型工程中——可以很好地保证函数接口的清晰性和正确性，并且始终高效。

concept_check 以库的方式提供了丰富的概念检查功能，用途非常广泛，只要我们编写泛型代码就可以使用概念检查库来约束模板参数，要求它必须满足某些要求。灵活使用概念检查，再结合 static_assert、type_traits 等其他工具可以有效地保证泛型代码的正确性。

function_types 是专门处理函数类型的 traits 工具，较 type_traits 库的 function_traits 处理的更加完整精细，不仅可以对函数元数据更精准的分类，还能够使用 mpl 容器执行分解和合成。

第 11 章

模板元编程（II）

在第 1 章中我们初步了解了模板元编程的知识，本章我们将更加深入地研究 C++中这一最为强大的编程武器，即 boost.mpl（meta programming library）。

mpl 是模板元编程的主要工具，它是整个 C++模板元编程的核心，也是理解 Boost 中许多其他组件工作原理的基础。熟悉并掌握 mpl 可以让我们把工作更多地放在编译期，更好地发挥 C++静态类型体系的优越性，开发出效率更高的运行时程序。

因为 mpl 十分庞大，故本书在这里只能择要介绍其中的部分内容，希望读者可以触类旁通。

11.1　mpl 概述

mpl 是专门为模板元编程创建的工具库，它以基本的元编程概念作为基础，辅以预处理元编程和其他几个基本库（如 type_traits），逐步发展出编译期的整数类型、类似 STL 的容器和算法，甚至还有编译期的 lambda 表达式，在 C++已有的语法体系中独立构造出了一个崭新的、完整的元编程体系。

mpl 提供了大量高质量高效率的元编程工具，它们大大降低了元编程的难度，使用这些高级元编程工具会使元编程工作更加的轻松和愉快，这些工具包括：

- 基本的数据类型　　：整数、pair、void 等运行时类型的对应物；
- 基本的数据运算　　：以元函数形式提供的算术运算、逻辑运算等运算功能；
- 程序流程控制　　：以元函数形式提供的分支流程处理功能；

- 容器　　　　　　　：模仿 STL 风格的存储元数据（类型）的编译期容器；
- 视图　　　　　　　：容器的适配器，可以简化容器的操作；
- 迭代器　　　　　　：模仿 STL 风格应用于容器的编译期迭代器；
- 算法　　　　　　　：模仿 STL 风格应用于容器和迭代器编译期算法；
- 高阶元数据　　　　：类似函数对象的元编程构件；
- bind 和 lambda 表达式：编译期的 lambda 表达式，功能强大灵活；
- 调试支持　　　　　：提供了编译期的断言和打印输出功能；
- 辅助宏　　　　　　：包括各种配置宏和 traits 宏。

mpl 是一个仅由头文件组成的库，无须编译，所有的组件均位于名字空间 boost::mpl，源代码则在目录<boost/mpl/>下，文件名与组件的名字通常是一致的。

由于不存在一个统一的头文件，所以使用元编程组件时必须手工添加包含的头文件，这样虽然比较麻烦，但也减少了不需要头文件的包含，一并减少了元计算（即编译）的时间，即：

```
#include <boost/mpl/xxx.hpp>          //所需的 mpl 元编程头文件
using namespace boost::mpl;           //打开名字空间
```

11.2 mpl 的整数类型

整数是任何编程方式都需要处理的数据，模板元编程当然也不例外，由于整数本身就是元数据，在编译期可以计算，所以元程序操作起来毫无困难。但直接操作整数对于元编程并不太方便，因为元编程更大的作用是类型计算，而整数并不是类型，所以直接使用整数计算不能够体现元编程类型计算的好处。

mpl 库为此提供了数值包装器概念，可以把整数（包括 int、long、bool 等类型）包装为值元函数（参见 1.1.2 小节），这样元程序就能够把整数用于类型计算，并基于这些高级整数类型建立起整个元编程计算体系。

11.2.1 概述

mpl 库里的数值包装器元函数共有六个，分别包装了如下不同的整数类型（注意没有

short)：

- char_<N> ：包装了 char 类型，值为 N；
- int_<N> ：包装了 int 类型，值为 N；
- long_<N> ：包装了 long 类型，值为 N；
- size_t<N> ：包装了 size_t 类型，值为 N；
- integral_c<T,N> ：包装了类型为 T 的整数类型，值为 N；
- bool_<N> ：包装了 bool 类型，值为 N。

这些元函数都具有基本相同的形式[①]，类摘要如下：

```
struct integral_wrapper                              //整数包装器
{
    BOOST_STATIC_CONSTANT(T, value = N);             //包装整数值 N
    typedef integral_c_tag tag;                      //类型标记
    typedef T type;                                  //返回自身
    typedef T value_type;                            //整数类型
    operator T() const { return this->value; }       //转型操作

    //bool_没有下列两个元数据
    typedef some_define next;                        //N++
    typedef some_define prior;                       //N--
};
```

数值包装器的功能比较简单，可以用::type 返回自身，用::value 返回被包装的整数值。对于非 bool 类型还提供了 next 和 prior 两个内部类型定义，可以获得类似 operator++和 operator--的效果，在元计算时递增或递减整数，另外我们也可以使用辅助元函数 next<>和 prior<>，效果相同但更加通用（参见 11.2.4 小节）。

数值包装器同时重载了转型操作，所以它们的实例可以在运行时隐式转换为被包装的整数，像真正的整数一样被使用。为了称呼方便，以下有时会将这些包装器元函数简称为“整数”或者“整数类型”，读者需自行辨析它在具体语境中的含义。

① 实际上除了 bool_，其他整数类型都是使用头文件<boost/mpl/aux_/integral_wrapper.hpp>进行宏替换实现的。

11.2.2 整数类型

整数类型的用法基本相同，所以在这里我们仅介绍 int_<>和 integral_c<>，这两个元函数的类摘要如下：

```
template<int N >
struct int_
{...};          //见上一小节 integral_wrapper 的代码

template<typename T, T N >
struct integral_c
{...};          //见上一小节 integral_wrapper 的代码
```

int_<>包装了标准的 int 类型，它使用::value 返回模板参数 N，而 integral_c<>因为支持任意整数类型所以有两个模板参数，使用::value 返回类型为 T 的模板参数 N。

示范整数类型包装器用法的代码如下：

```
#include <boost/type_traits.hpp>                              //用于类型比较
#include <boost/mpl/integral_c.hpp>                           //任意整数包装器
#include <boost/mpl/int.hpp>                                  //int 包装器
#include <boost/mpl/next_prior.hpp>                           //递增递减元函数
using namespace boost;
using namespace boost::mpl;

int main()
{
  typedef int_<2> i2;                                         //包装 int 型整数 2
  typedef integral_c<short, 2> s2;                            //包装 short 型整数 2

  assert(i2::value == 2);                                     //获取整数值
  assert(i2::value == s2::value);                             //两个类型的值相等

  assert((is_same<i2::type, i2>::value));                     //返回包装器自身
  assert((is_same<s2::value_type, short>::value));            //返回整数类型

  assert(i2::next::value == 3);                               //内部成员获得递增值
  assert(prior<s2>::type::value == 1);                        //使用元函数获得递减值

  i2 two1;                                                    //声明两个包装器实例
```

```
    s2 two2;                        //值均为常量 2

    int i = two1 + two2;            //隐式类型转换为 int 参与运行时计算
    assert(i == int_<4>());         //与 int_<>元函数比较，使用()创建临时对象
}
```

这段示例代码中我们混合了编译期的元编程和运行时的普通编程。需要注意的是编译期包装器不能直接与整数进行比较，必须使用::value 获得整数值才能与整数比较。元函数 next<>/prior<>返回的是包装器，所以要先用::type 获得元函数返回值才能调用::value。运行时由于包装器有隐式类型转换，所以实例可以直接参与整数运算。

11.2.3 bool 类型

bool 包装器 bool_<>是一个比较特殊的元函数，因为它的取值只有 true/false 两个值，而且也不能递增或递减，它的类摘要如下：

```
template< bool C_ >
struct bool_
{
    BOOST_STATIC_CONSTANT(bool, value = C_);            //包装 bool 值 C_
    typedef integral_c_tag tag;                         //类型标记
    typedef bool_ type;                                 //返回自身
    typedef bool value_type;                            //bool 类型
    operator bool() const { return this->value; }       //转型操作
};
```

bool_<>的用法与 int_<>和 integral_c<>差不多甚至更加简单，只是需要注意没有 next/prior 成员，也不能使用 next<>/prior<>元函数。为了方便使用，mpl 还提供了两个快捷 typedef：

```
typedef bool_<true>  true_;
typedef bool_<false> false_;
```

示范 bool_<>用法的代码如下：

```
#include <boost/mpl/bool.hpp>                           //bool 包装器
#include <boost/mpl/next_prior.hpp>
using namespace boost;
using namespace boost::mpl;
```

```
int main()
{
  assert(true_::value == true);
  assert(false_::value == false);

  assert((is_same<true_::type, bool_<true> >::value));
  assert((is_same<false_::value_type, bool>::value));

  next<true_>::type;                          //不能执行递增操作，编译错误
}
```

bool_<>在本书中的序列化库 serialization 有具体应用，参见 9.4 小节。

11.2.4 基本运算

同 C++内建的整数运算支持一样，mpl 库也对这些元编程整数类型提供了等价的编译期运算支持——当然，对这些整数类型的运算只能使用元函数。

mpl 库对整数类型运算的支持是非常全面的，不仅有我们之前看到的递增递减运算，还包括算术运算、比较运算等，详细如下：

- 递增递减运算：位于头文件<boost/mpl/next_prior.hpp>，包括 next<>和 prior<>两个元函数；
- 算术运算：位于头文件<boost/mpl/arithmetic.hpp>，包括加减乘除、取模、取负值，元函数如 plus<>、minus<>；
- 比较运算：位于头文件<boost/mpl/comparison.hpp>，包括小于、大于、等于和不等于共 6 种，元函数如 less<>、greater<>；
- 位运算：位于头文件<boost/mpl/bitwise.hpp>，包括与、或、异或、移位等，元函数如 bitand_、bitor_；
- 逻辑运算：位于头文件<boost/mpl/logical.hpp>，包括与、或、非 3 种逻辑运算，元函数如 and_、or_和 not_；
- 其他运算：包括最大最小值 min<>/max<>、整数的大小 sizeof_<>等。

可以看到 mpl 为模板元编程提供了完整的整数运算能力，能够在编译期执行任何整数计算工作。

由于整数运算元函数很多，不可能一一介绍，故下面仅列举一些较常用的运算元函数。

递增递减运算

递增递减运算元函数包括 next<>和 prior<>，它们相当于运行时的 operator++和 operator--，可以对整数执行递增递减操作，以::type 返回运算后的整数，简化的类摘要如下：

```
template<T>
struct next
{
    typedef typename T::next type;    //返回 T 的 next 成员
};

template<T>
struct prior
{
    typedef typename T::prior type;   //返回 T 的 prior 成员
};
```

从 next<>和 prior<>的实现代码可以看出，它们只能应用于非 bool 类型的整数，因为只有非 bool 类型的整数才有 next 和 prior 成员。

算术运算

算术运算元函数包括加减乘除、取模和取负值六个元函数，支持使用多个参数参与运算（默认最多五个），其中常用的几个摘要如下：

```
template<typename T1, typename T2, ...>
struct plus                           //加法运算
{
    typedef some_define type;         //返回运算后的整数
};

template<typename T1, typename T2, ...>
struct minus                          //减法运算
{
    typedef some_define type;         //返回运算后的整数
};
```

```
template<typename T>
struct negate                           //取负值运算
{
    typedef some_define type;           //返回运算后的整数
};
```

比较运算

比较运算包括小于、小于等于、大于、大于等于、等于和不等于共六个元函数，它们都只有两个模板参数，返回 bool_<>类型，名字与标准库的比较函数对象相同，常用的几个摘要如下：

```
template<typename T1, typename T2>
struct less                             //小于关系比较
{
    typedef some_define type;           //返回 bool_<>
};

template<typename T1, typename T2>
struct greater                          //大于关系比较
{
    typedef some_define type;           //返回 bool_<>
};

template<typename T1, typename T2>
struct equal_to                         //等于关系比较
{
    typedef some_define type;           //返回 bool_<>
};
```

逻辑运算

逻辑运算包括与或非三个元函数，返回 bool_<>类型，其中的与或运算支持多个模板参数，摘要如下：

```
template<typename T1, typename T2, ...>
struct and_                             //逻辑与运算
{
    typedef some_define type;           //返回 bool_<>
};
```

```
template<typename T1, typename T2, ...>
struct or_                          //逻辑或运算
{
   typedef some_define type;        //返回 bool_<>
};

template<typename T>
struct not_                         //逻辑非运算
{
   typedef some_define type;        //返回 bool_<>
};
```

示例代码

示范这些整数运算元函数的代码如下：

```
#include <boost/mpl/bool.hpp>
#include <boost/mpl/int.hpp>
#include <boost/mpl/arithmetic.hpp>
#include <boost/mpl/logical.hpp>
#include <boost/mpl/comparison.hpp>
using namespace boost;
using namespace boost::mpl;

int main()
{
  typedef int_<2> i2;                                           //定义 3 个整数类型
  typedef int_<5> i5;
  typedef int_<7> i7;

  assert((plus<i2, i5, i7>::type::value == 14));                //加法
  assert((equal_to<minus<i7, i5>::type, i2>::type::value)); //减法

  assert((less<i2, i7>::type::value));                          //小于比较
  assert((is_same<greater<i5, i2>::type, true_>::value));     //大于比较

  assert((not_<and_<true_, false_>::type>::type::value));     //逻辑与
  assert((or_<true_, false_>::type()));          //逻辑或，使用了隐式类型转换
}
```

这段代码比较简单，读者需要注意减法运算中我们使用的是相等比较 equal_to<>元函

数，而不是之前我们常用的 is_same<>。这是因为整数的算术运算后的结果不一定是与参数同类型的整数，因为有可能混用不同的整数类型，所以结果通常都是一个 integral_c<>类型，只能使用 equal_to<>来比较数值的等价，例如：

```
assert(!(is_same<plus<int_<1>, char_<2>, long_<3>>::type,
  long_<6>>::type::value));            //混用 int_<>和 long_<>类型
assert((is_same<plus<int_<1>, char_<2>, long_<3>>::type,
  integral_c<long, 6>>::type::value));   //结果是 integral_c<>类型
```

11.3 mpl 的流程控制

元程序本质上也是程序，它也具有顺序、分支和循环三类程序结构。但它不同于普通的运行时程序，属于函数式编程，没有循环和跳转语句，最常用的循环手段是递归然后模板特化终止。

对于分支结构，mpl 提供了四个元函数，它们类似运行时的 if-else 语句，我们已经在 1.2.9 小节见到了它们的初步用法。

11.3.1 if_和 if_c

if_<>和 if_c<>位于头文件<boost/mpl/if.hpp>，是两个最简单的分支元函数，可以近似地看做是?:操作符，能够像 if-else 语句一样根据条件执行不同的分支，类摘要如下：

```
template<bool C, typename T1, typename T2>
struct if_c                                  //if_c<>元函数标准形式
{
   typedef T1 type;                          //C 为 true 返回第一个元数据
};
template<typename T1, typename T2>
struct if_c<false,T1,T2>                     //if_c<>对 false 模板偏特化
{
   typedef T2 type;                          //C 为 false 返回第二个元数据
};

template<typename C, typename T1, typename T2>
struct if_                                   //if_<>元函数
{
```

```
    typedef if_c<C::value, T1, T2> almost_type_; //调用 if_c<>元函数
    typedef typename almost_type_::type type;
};
```

if_c<>和 if_<>非常相似，不同的是 if_c<>的条件是一个 bool 类型，而 if_<>的条件是一个有::value 返回的值元函数（不一定是 bool_<>）。两者的应用范围各有不同，对于 type_traits、mpl 等库中的大量元函数来说，if_<>因为可以少写一个::value 会更方便一些。

简单示范 if_c<>和 if_<>用法的代码如下：

```
#include <boost/mpl/if.hpp>
using namespace boost;
using namespace boost::mpl;

int main()
{
  typedef if_c<true, int, long>::type mdata1;          //使用 if_c<>计算
  assert((is_same<mdata1, int>::value));              //得到元数据 int

  typedef if_<false_, float, double>::type mdata2;    //使用 if_<>计算
  assert((is_same<mdata2, double>::value));           //得到元数据 double

  typedef if_<is_integral<mdata2>,                  //使用 is_integral<>作为条件
      integral_promotion<mdata2>::type,               //提升整数类型
      floating_point_promotion<mdata2>::type          //提升浮点数类型
  >::type mdata3;
  assert((is_same<mdata3, double>::value));           //得到元数据 double
}
```

if_c<>和 if_<>并非全无缺点，因为它们在元计算时必须计算出所有的元数据，而不能根据条件有选择地“忽略”不需要计算的数据（即缓式评估 lazy evaluation），增加了元计算的时间。

11.3.2 eval_if 和 eval_if_c

eval_if<>和 eval_if_c<>位于头文件<boost/mpl/eval_if.hpp>，解决了 if_c<>和 if_<>不能缓式评估的缺点，它们专门用于计算元函数，可以根据条件只计算需要的部分，类摘要如下：

```
template<typename C, typename F1, typename F2>
struct eval_if                              // eval_if<>元函数
    : if_<C,F1,F2>::type                    //元函数转发给 if_<>，并得到返回结果
{};

template<bool C, typename F1, typename F2>
struct eval_if_c                            // eval_if_c<>元函数
    : if_c<C,F1,F2>::type                   //元函数转发给 if_c<>，并得到返回结果
{};
```

eval_if<>和 eval_if_c<>的实现代码非常简单，注意它们不仅仅是元函数转发，同时还使用::type 得到了 if_c<>和 if_<>的计算结果（即 F1 或 F2），所以使用 eval_if<>::type 或 eval_if_c<>::type 就相当于 F1::type 或 F2::type，方便了很多。

eval_if<>和 eval_if_c<>的后两个模板参数必须是元函数，在使用的时候需要特别注意，它们名字中的“eval”清楚地表明了这一点。

eval_if<>和 eval_if_c<>示范代码如下，与上一小节的很相似：

```
#include <boost/mpl/eval_if.hpp>
#include <boost/mpl/identity.hpp>
using namespace boost;
using namespace boost::mpl;

int main()
{
  typedef eval_if_c<true,                   //使用 eval_if_c<>计算
      identity<int>, identity<long>>::type mdata1;
  assert((is_same<mdata1, int>::value));

  typedef eval_if<false_,                   //使用 eval_if <>计算
      identity<float>, identity<double>>::type mdata2;
  assert((is_same<mdata2, double>::value));

  typedef eval_if<is_integral<mdata2>,      //使用 is_integral<>作为条件
      integral_promotion<mdata2>,
      floating_point_promotion<mdata2>
  >::type mdata3;
  assert((is_same<mdata3, double>::value));
}
```

这段代码中我们使用了另外的一个元函数 identity<>，它是一个很小的辅助元函数，直接返回参数自身，很类似函数对象 identity（参见 7.4.2 小节）。

11.4 mpl 的容器

作为一个完整的模板元编程框架，mpl 不仅拥有整数类型和流程控制，C++标准库中的容器也有对应的元编程版本，这就是 mpl 容器。

11.4.1 概述

mpl 中的容器与 STL 中的容器很相似，所以可以把它们对比研究，这样更容易学习[①]。

mpl 容器可分为序列容器和关联容器两类，当然，容器里容纳的元素都是元数据——也就是类型，所以没有 STL 容器对元素的可拷贝可赋值的要求，并且元素可以是任意类型（有点类似 tuple）。mpl 容器可以添加删除元素，也有迭代器的概念，也可以在这些容器上使用算法，这些都与 STL 容器很相像。但 mpl 容器与 STL 容器的一个重要区别是它们虽然也是类型（元数据），但没有成员函数（这是运行时的概念），自身没有操纵容器内元素的能力，只能通过外部元函数才能处理元数据。

mpl 提供几乎与 STL 容器完全等价的序列容器和关联容器，它们同时也是元函数，能够以::type 返回自身。

mpl 序列容器包括：

- list ：不同于 std::list，它是一个单向链表，只能在序列的前端操作元素，可以容纳无限个元素；
- vector：它非常像 std::vector，是一个可以随机访问的序列，但它有具有一些 std::list 的特性，可以在序列的两端操作元素。另外不同于 std::vector 的一点是它的容量是有限的，不能无限增加，缺省最多能够容纳 20 个元数据；
- deque ：它类似 std::deque，可以在序列两端操作元素，几乎与 vector 是相同的；

mpl 关联容器缺省最多能够容纳 20 个元素，包括：

① 在 mpl 库中把容器称为“序列”（sequence），并且有自己的概念、分类和定义，与 STL 相似但也有不同，本书为了方便学习沿用了标准库的描述方式，不涉及较复杂的概念。

- set：类似 std::multiset，是一个允许重复的元数据集合；
- map：类似 std::multimap，是一个允许重复的元数据映射关系集合。

除了以上五个基本容器，mpl 库还针对整数类型（数值包装器）提供了一些特别的整数容器：

- range_c<T,N,M> ：不可修改的包含 [N,M) 区间内整数的 vector 容器；
- list_c<T,...> ：包含类型为 T 的若干个整数的 list 容器；
- vector_c<T,...> ：包含类型为 T 的若干个整数的 vector 容器；
- set_c<T,...> ：包含类型为 T 的若干个整数的 set 容器；
- string ：专门存储 char 字符的容器，类似 std::string，可以在编译期处理的字符串。

判断一个类型是否是 mpl 容器可以使用值元函数 is_sequence<>，例如：

```
assert(!is_sequence<int>::value);
assert((is_sequence<mpl::vector<>>::value));
assert((is_sequence<mpl::set<int,char>>::value));
```

由于 mpl 容器较多，而且实现较复杂，我们只介绍较为常用的 vector、string 和 map，mpl 的容器之上的视图（view）概念本书不做介绍。

11.4.2 vector

vector 容器很像是 boost::tuple 的编译期版本，可以容纳异种类型（元数据），是最容易使用的 mpl 容器。

由于 vector 使用了预处理元编程，它的实现代码很难直接给出，示意形式如下：

```
template< t1, t2,...>                     //容纳若干个元数据
struct vector{...};                       //具体实现代码无法给出
```

我们可以在编译期使用相关的元函数处理 vector 里的元数据，如判断容器的大小、获取前端和末端的元素、添加或者删除元素等。示范 vector 用法的代码如下，其中涉及的元函数参见 11.4.5 小节：

```
#include <boost/mpl/vector.hpp>                              //vector 头文件
#include <boost/mpl/empty.hpp>                               //各个操作元函数的头文件
#include <boost/mpl/size.hpp>
#include <boost/mpl/front.hpp>
#include <boost/mpl/back.hpp>
#include <boost/mpl/at.hpp>
#include <boost/mpl/push_front.hpp>
#include <boost/mpl/clear.hpp>
using namespace boost::mpl;

int main()
{
  typedef mpl::vector<char, int , long,                      //容纳五个元数据的 vector
      int_<8>, std::string> vec;                             //可以不必使用::type

  assert((empty<vec>::value == false));                         //判断容器不空
  assert((size<vec>::value == 5));                              //检查容器的大小

  assert((is_same<front<vec>::type, char>::value));             //访问前端元素
  assert((is_same<back<vec>::type, std::string>::value));//访问末端元素
  assert((is_same<at_c<vec, 1>::type, int>::value));            //随机访问元素

  typedef push_front<vec, float>::type vec2;                    //前端添加元素 float
  assert((size<vec2>::value == 6));                             //元素数量加 1
  assert((is_same<front<vec2>::type, float>::value));           //访问前端元素

  typedef clear<vec2>::type vec3;                               //清空容器
  assert((empty<vec3>::value));                                 //判断容器已经被清空
}
```

从这段代码中我们可以看到 vector 的用法与 std::vector 的用法非常相似，只是它容纳的是 C++的类型元数据，必须在编译期使用元函数以自由函数而不是成员函数的方式操作。另外还需要注意的是因为容器是元数据，而元数据是不可变的，所以对容器的任何变动操作都必须返回一个新的容器，原容器不会变化。

11.4.3 string

string 是一类特殊的 mpl 序列容器，它专门容纳 char 类型的字符，相当于字符串常量的元编程版本，它的示意形式如下：

```
template<c1, c2, ...>                     //容纳若干个字符数据
struct string{...};                       //具体实现代码无法给出
```

string 的模板参数较为特别，它们可以是任意多组使用单引号（注意!）的 char 型字面量，由于受到预处理的限制每组字符最多只能有四个字符，因此如果要容纳一个较长的字符串就必须适当分组，例如：

```
typedef mpl::string<'a','b','c'> abc;     //字符串是"abc"
typedef mpl::string<'Hell','o !!'> hello;//字符串是"Hello !!"
typedef mpl::string<'Hello !!'> hello2;   //编译错误，字符过多
typedef mpl::string<"Hello !!"> hello3;   //编译错误，使用了双引号
```

string 容纳的字符串长度也是有限制的，mpl 缺省配置最大能够容纳 32 个字符，如果需要可以在<boost/mpl/string.hpp>前使用宏 BOOST_MPL_LIMIT_STRING_SIZE 更改。

string 具有同 vector 一样的能力，也可以判断字符串的大小、获取前端和末端的元素、添加或者删除元素，这里的元素都是 char_<>类型。除了这些操作之外，string 还有一个特别的值元函数 c_str<>，它可以用::value 返回一个 NULL 结尾的标准字符串，可以在运行时使用。

示范 string 用法的代码如下：

```
#define BOOST_MPL_LIMIT_STRING_SIZE 24                  //允许最多 24 个字符
#include <boost/mpl/string.hpp>                         //string 头文件
using namespace boost::mpl;

int main()
{
  typedef mpl::string<'Hell','o !!'> hello;            //一个字符串容器

  assert((empty<hello>::value == false));              //判断非空
  assert((size<hello>::value == 8));                   //字符串长度为 8

  assert((front<hello>::type::value == 'H'));          //取第一个字符
  assert((back<hello>::type::value == '!'));           //取最后一个字符

  typedef push_back<hello, char_<'?'>>::type hello2; //追加一个字符
  assert((c_str<hello2>::value ==                    //使用 c_str<>获得编译器字符串常量
      std::string("Hello !!?") ));                   //与运行时 std::string 比较
}
```

11.4.4 map

map 本质上与 vector、string 差不多，都是容纳元数据的容器，但它的元素类型却比较特别，是一个 mpl::pair 类型，里面包含了键-值的映射关系。

mpl::pair 很像 std::pair，不同的是它把元数据作为成员，定义如下：

```
template<typename T1, typename T2>
struct pair
{
    typedef pair type;                    //返回自身
    typedef T1 first;                     //第一个成员
    typedef T2 second;                    //第二个成员
};
```

因为 pair 是一个 struct，所以我们可以使用::first 或::second 直接获得其中的两个成员，也可以使用元函数 first<>和 second<>间接获得。

map 使用 mpl::pair 作为元素，示意代码如下：

```
template<p1, p2, ...>                     //容纳若干个键-值对元数据
struct map{...};                          //具体实现代码无法给出
```

map 具有大多数容器的共通操作，如判断容器的大小、添加或者删除元素等，但它只能获取前端元素，没有操纵末端元素的 back<>元函数，此外还有一些专门用于键-值操作的专用元函数，如 has_key<>、count<>等。

示范 map 用法的代码如下：

```
#include <boost/mpl/map.hpp>              //map 头文件
#include <boost/mpl/count.hpp>            //count 算法
using namespace boost::mpl;

int main()
{
  typedef mpl::map<                       //定义 map 容器
      mpl::pair<int, std::vector<int>>,   //以下是三个 pair
      mpl::pair<char,std::string>,        //元数据可以任意映射
      mpl::pair<int_<3>, float[3]>        //一个数值包装器映射到一个浮点数组
  > m;                                    //map 容器名字是 m
```

```
    assert((empty<m>::value == false));                    //判断非空
    assert((size<m>::value == 3));                         //容器大小为 3

    typedef front<m>::type p1;                             //获取第一个元素
    assert((is_same<first<p1>::type, int>::value));        //使用 first<>元函数
    assert((is_same<p1::second,std::vector<int>>::value)); //直接取成员

    assert((has_key<m, char>::value));                 //检查键是否存在
    assert((count<m, int_<0>>::value == 0));           //count 算法计算键的个数

    typedef at<m, char>::type mdata;                   //根据键获得值
    assert((is_same<mdata, std::string>::value));
}
```

11.4.5 相关元函数

mpl 为容器提供了数个元函数，它们的功能基本与 STL 的同名成员函数等价，常用的元函数可以分类如下。

容器容量操作：

- empty<C> ：值元函数，检查容器是否为空；
- size<C> ：值元函数，获得容器的大小。

元素访问操作：

- front<C> ：返回容器里的第一个元素；
- back<C> ：返回容器里的最后一个元素，对于 list 和关联容器不适用；
- at<C,k> ：对于序列容器，返回第 k 个位置上的元素（k 是个数值包装器），对于关联容器，返回键为 k 的元素；
- at_c<C,n>：仅适用于序列容器，返回第 n 个位置上的元素（n 是个 long 型整数）。

迭代器操作（参见 11.5 小节）：

- begin<C> ：返回一个容器对应的迭代器，指向第一个元素的位置；
- end<C> ：返回一个容器对应的迭代器，指向最后一个元素之后的位置（逾尾）。

元素变动操作：

- push_front<C,t> ：在容器前端插入一个元素，返回新容器；
- push_back<C,t> ：在容器末端插入一个元素，返回新容器；
- pop_front<C> ：删除容器的第一个元素，返回新容器；
- pop_back<C> ：删除容器的最后一个元素，返回新容器；
- clear<C> ：清除容器里的所有元素，返回新容器；
- erase<C,pos>/erase<C,f,l> ：删除容器中某个位置或者某个区间里的元素，其中的 pos、f 和 l 都是容器的迭代器；
- insert<C,pos,t>/insert<C,t>：向序列容器的 pos 位置或关联容器里插入一个元素，其中的 pos 是容器的迭代器。

关联容器特有操作：

- has_key<C,k> ：检查容器中是否有键为 k 的元素；
- erase_key<C,k>：删除容器中键为 k 的元素。

这些元函数与标准库的容器操作很类似，用法也比较简单，示范代码可见前几节对容器的操作示例。

11.5 mpl 的迭代器

在 mpl 中迭代器同样占有非常重要的地位，它是 mpl 容器和 mpl 算法之间的粘接剂，本节我们简要研究 mpl 中的元编程迭代器，读者可与第 3 章对比，以作参考。

11.5.1 概述

mpl 中的迭代器也遵循迭代器设计模式，与 STL 标准迭代器和 Boost 新式迭代器不同的地方仅在于它是编译期的迭代器，只能使用元函数操纵。

mpl 迭代器具有基本的操作接口，包括解引用、前进、后退、计算距离和判等。它也能够按照取值和遍历操作进行分类，但因为元数据都是不可变的，所以所有的 mpl 都是只读迭代器，故我们仅需以遍历方式进行分类。

mpl 迭代器可分为以下三类，与 Boost 新式迭代器颇类似但要简单：

- 前向迭代器　　：可以使用元函数 next<>递增；
- 双向迭代器　　：在前向迭代器的基础上增加使用元函数 prior<>递减，也就是可以逆向遍历；
- 随机访问迭代器：在双向迭代器的基础上增加迭代器的距离运算。

有了迭代器的概念，mpl 容器也可以按照提供的迭代器进行分类[1]，具体分类如下：

- 前向序列　　：所有的容器都符合前向序列的概念，其迭代器可以执行递增操作；
- 双向序列　　：string、vector 和 deque 符合双向序列的概念；
- 随机访问序列：vector 和 deque 符合随机访问序列的概念。

11.5.2　相关元函数

与运行时的迭代器一样，mpl 迭代器也可以执行解引用、前进、后退等操作，只不过这些操作返回的都是编译期的元数据。

mpl 提供的迭代器元函数有：

- deref<I>　　：解引用迭代器，返回元数据；
- next<I>　　：递增迭代器；
- prior<I>　　：递减迭代器；
- advance<I,n>　　：迭代器移动 n 个位置，对于双向迭代器 n 可以是负值；
- distance<I1,I2>　　：返回两个迭代器之间的距离；
- iterator_category<I>：获得迭代器的分类标志。

关于元函数 next<>/prior<>要做一点特别的说明，它们实际上与 11.2 小节操作整数的 next<>/prior<>是完全相同的。这是由于泛型编程的静态多态特性，只要元数据有内部的::next/::prior 定义，对于整数和迭代器就可以分别执行不同的操作。

mpl 的迭代器通常可以用容器的 begin<>和 end<>元函数获得，示范用法的代码如下：

---

① mpl 中容器的分类还有关联性和可扩展性两个概念，本书从略。

```
#include <boost/mpl/vector.hpp>                    //vector 容器
#include <boost/mpl/begin_end.hpp>                 //各种迭代器操作函数
#include <boost/mpl/next_prior.hpp>
#include <boost/mpl/deref.hpp>
#include <boost/mpl/advance.hpp>
#include <boost/mpl/distance.hpp>
using namespace boost::mpl;

int main()
{
  typedef mpl::vector<char, int , long,            //容纳 5 个元数据的 vector
      int_<8>, std::string> vec;

  typedef begin<vec>::type i1;                     //获得随机访问迭代器
  assert((is_same<deref<i1>::type, char>::value)); //解引用迭代器

  typedef next<i1>::type i2;                       //递增迭代器
  assert((is_same<deref<i2>::type, int>::value));  //解引用迭代器
  assert((is_same<i1, prior<i2>::type>::value));   //递减迭代器并比较

  typedef mpl::advance<i2, int_<4>>::type i3;      //前进迭代器至末尾
  assert((mpl::distance<i1, i3>::type::value == 5));//计算迭代器间距离
  assert((mpl::distance<i3, i1>::type::value == -5));
  assert((is_same<i3, end<vec>::type>::value));    //与逾尾迭代器比较

  typedef insert<vec, i2, float>::type vec2;       //在 i2 的位置插入一个元数据
}
```

11.6 mpl 的算法

mpl 同 STL 一样基于编译期的容器和迭代器提供了大量的算法，使得我们可以对存储在容器中的元数据执行复杂的计算，许多算法与 STL 是对应的。

由于 mpl 中的算法很多，本节仅择要介绍一些较常用的算法，另一些算法参见 11.7.5 小节。

11.6.1 插入器

mpl 中的插入器（inserter）类似 STL 中的插入迭代器的概念，它可以把一个容器适配成迭代器供算法操作。

mpl 插入器共有以下三个：

- back_inserter<C> ：在容器后端插入元素，要求容器支持 push_back<>；
- front_inserter<C>：在容器前端插入元素，要求容器支持 push_front<>；
- inserter<S,Op> ：通用的插入器，以一个初始状态 S 开始执行 Op 操作完成插入动作。

这三个插入器中最常用的是前两个 front_inserter<>和 back_inserter<>，而第三个 inserter<>通常需要配合元编程 lambda 表达式使用（参见 11.7.5 小节）。

示范插入器用法的代码如下：

```
#include <boost/mpl/vector.hpp>
#include <boost/mpl/string.hpp>
#include <boost/mpl/back_inserter.hpp>      //后端插入器
#include <boost/mpl/front_inserter.hpp>     //前端插入器
#include <boost/mpl/copy.hpp>               //copy 算法
#include <boost/mpl/size.hpp>
using namespace boost::mpl;

int main()
{
  typedef mpl::vector<char, int , long> vec; //容纳三个元素

  typedef mpl::copy<vec,
      mpl::back_inserter<vec>>::type vec2;   //插入自己，相当于双倍
  assert((size<vec2>::value == 6));

  typedef mpl::string<> str1;               //编译期空字符串
  typedef mpl::copy<
      mpl::string<'time'>,                  //使用一个“临时”元数据
      mpl::front_inserter<str1>>::type str2; //前端插入
  cout << c_str<str2>::value;               //输出字符串 emit
}
```

11.6.2 查询算法

查询算法也可以称为只读算法，因为它们只是以只读的方式访问容器里的元素，不会变

动容器。这些算法与标准算法非常相似，常用的有如下几个：

- find<C,t>　　　：在容器中插入元素 t，返回迭代器；
- contains<C,t>：值元函数，检查容器中是否存在元素 t；
- count<C,t>　　：值元函数，返回容器中元素 t 的个数；
- equal<C1,C2>　：值元函数，比较两个容器是否等价，即元数据相同。

我们需要注意算法 equal<>，不同于 type_traits 库的元函数 is_same<>，它比较的是两个容器的等价性，两个等价的 mpl 容器完全有可能是两个不同的类型，所以比较容器时通常应该使用 equal<>算法而不是 is_same<>元函数。

示范这些算法的代码如下：

```
#include <boost/mpl/vector_c.hpp>
#include <boost/mpl/range_c.hpp>
#include <boost/mpl/find.hpp>
#include <boost/mpl/count.hpp>
#include <boost/mpl/contains.hpp>
#include <boost/mpl/equal.hpp>
#include <boost/mpl/size.hpp>
using namespace boost::mpl;

int main()
{
  typedef range_c<int, 0, 10> rc;                //定义一个整数区间
  assert((size<rc>::value == 10));

  //使用宏简化容器内元素的写法
  #define INTV(n) integral_c<rc::value_type, n>

  assert((deref<                                  //解引用迭代器或者元素
      find<rc, INTV(0)>::type                     //查找容器内的元素
      >::type::value == 0));

  assert((contains<rc, INTV(9)>::value));         //检查元素是否存在
  assert((!contains<rc, INTV(10)>::value));       //检查元素是否存在

  assert((count<rc, INTV(5)>::value == 1));       //计算元素的数量
```

```
    assert((count<rc, INTV(-5)>::value == 0));         //计算元素的数量

    typedef vector_c<int, 0,1,2,3,4,5,6,7,8,9> vec;    //定义一个整数容器
    assert((mpl::equal<rc, vec>::value));              //等价比较
    assert((!is_same<rc, vec>::value));                //两者是不同的类型
}
```

这段代码中我们定义了一个宏 INTV 用来简化代码的熟悉，这是因为 range_c 等整数容器内部存储的元素都是 integral_c，而不是 int_、long_等类型，所以在查询元素时必须使用相同类型的元素，如果直接使用 int_等类型会发生编译错误。

11.6.3 变换算法

变换算法处理容器中的全部或部分元素，然后返回一个新的容器。

mpl 中的部分变换算法如下，并且提供带前缀“reverse_”的形式可以返回操作后的逆序容器：

- copy<From,To> ：拷贝一个容器里的所有元素；
- replace<From,Old,New,To>：把容器中的 Old 元素全部替换为 New 元素；
- remove<From,t,To> ：移除容器中所有值为 t 的元素；
- reverse<From,To> ：逆序拷贝容器里的所有元素，相当于 reverse_copy。

这四个算法都比较简单，需要注意的是算法中的最后一个参数 To，它可以省略不用。如果不使用 To 参数，那么算法直接用::type 返回变换后的新容器，如果使用 To 参数，通常需要搭配插入器（参见 11.6.1 小节）工作。

示范算法的代码如下：

```
#include <boost/mpl/vector.hpp>
#include <boost/mpl/reverse.hpp>
#include <boost/mpl/replace.hpp>
#include <boost/mpl/remove.hpp>
using namespace boost::mpl;

int main()
{
    typedef mpl::vector<char, short, int, long> vec;   //容纳 4 个元素
```

```
    typedef mpl::replace<vec,                           //替换 vec 中的元素
        int, char>::type vec2;                          //int 替换为 char

    assert((mpl::count<vec2, char>::value == 2));       //容器里有两个 char
    assert((is_same<deref<                    //使用迭代器取第三个位置的元素比较验证
                mpl::advance<begin<vec2>::type, int_<2>>::type>::type,
                char>::value));

    typedef mpl::remove<vec2, short>::type vec3;        //删除 short 元素
    assert((contains<vec3, short>::value));             //容器中不存在 short 元素

    assert((mpl::equal<                                 //比较等价性
        mpl::reverse<vec3>::type,                       //逆序拷贝
        mpl::vector<long, char, char>                   //一个“临时”容器
        >::value));
}
```

11.6.4 运行时算法

mpl 库中有一个特别的 for_each 算法，它工作在**运行时**，可以遍历类型容器，调用一个函数对象操作类型容器里类型对应的实例对象。

for_each 算法是一个模板函数，声明如下：

```
template<typename Sequence, typename F>
void for_each(F f);
```

for_each 算法有两个模板参数：第一个参数 Sequence 是 mpl 容器，必须显式指定；第二个参数 F 是一个函数对象，它应该具有模板成员函数 operator()，能够处理 Sequence 中的所有类型，否则会发生编译错误。

为了示范 for_each 算法的用法，我们先定义两个函数对象：

```
struct mpl_func1
{
  template<typename T>
  void operator()(T t)              //输出类型名
  {   cout << typeid(t).name() << endl;   }
};
```

```
struct mpl_func2
{
  template<typename T>
  void operator()(T t)                          //输出整数类型的值
  {
      if (is_same<tag<T>::type, integral_c_tag>::value)
      {   cout << t << ',';}
  }
};
```

第一个函数对象 mpl_func1 很简单，仅仅是使用 typeid 得到类型的名称，第二个函数对象 mpl_func2 略复杂一些，它使用了元函数 tag<>获得了类型的标志，仅当类型是一个 mpl 整数时才输出。

示范 for_each 算法用法的代码如下：

```
#include <boost/mpl/for_each.hpp>
using namespace boost::mpl;

int main()
{
  typedef range_c<int, 0, 5> rc;                //定义一个整数范围

  typedef mpl::vector<> vec;                    //一个空类型容器

  typedef mpl::copy<rc,                         //copy 算法拷贝整数到空容器
      mpl::back_inserter<vec>>::type vec2;

  typedef push_front<vec2, float>::type vec3;   //容器前端添加一个元素

  mpl::for_each<vec3>(mpl_func1());             //输出容器中的类型信息

  cout << endl;

  mpl::for_each<vec3>(mpl_func2());             //输出容器中的整数
}
```

代码的运行结果如下：

```
float
struct boost::mpl::integral_c<int,0>
```

```
struct boost::mpl::integral_c<int,1>
struct boost::mpl::integral_c<int,2>
struct boost::mpl::integral_c<int,3>
struct boost::mpl::integral_c<int,4>
0,1,2,3,4,
```

11.7 mpl 的高级用法

本节我们简要介绍 mpl 库中的一些高级特性，主要是用于参数绑定的 bind 表达式和 lambda 表达式。

bind 和 lambda 表达式都是函数式编程中的重要角色，它们强化了函数对象的作用，STL 有简单的 bind1st/bind2nd，Boost 库提供了运行时的 bind 和 lambda 表达式库，C++11 标准也定义了语言级别的 lambda 表达式，而在 mpl 中则提供了编译期的 bind 和 lambda 表达式，配合算法使用会令模板元编程更加强大（和复杂）。

11.7.1 高阶元数据

在研究编译期 bind 和 lambda 表达式之前，我们先来了解一个元编程新概念：高阶元数据[①]。

高阶元数据是一种特殊的类元数据，它内嵌有名一个为 apply<>的元函数[②]，形如：

```
struct high_order_meta_data
{
  template<typename T1, typename T2, ...>
  struct apply
  {
      typedef some_define type;
  };
};
```

高阶元数据主要的功能是包装元函数，把它变成元数据，从而可以把元函数传递给其他

① 高阶元数据这个名词是作者在实践中自行“发明”的，目前现有的元编程资料并没有使用这个词来称呼这种元编程构件，在 Boost 文档中使用的名称是“元函数类”。但作者个人认为这个名词欠妥，因为在模板元编程中已经不存在 C++的 class 概念了，所有的类型（type）都作为元数据出现，使用“类”这个称呼容易造成概念上的混乱。

② 高阶元数据内部的 apply<>元函数命名只是 mpl 的约定，我们也可以自行选择其他的名字，例如 6.2.6 小节就使用了 pack<>。

的元函数进行调用。从功能来看，它的作用颇类似函数对象（function object），元函数 apply<>可以看做是运行时的 operator()。

与高阶元数据对应，操作或者返回高阶元数据的元函数就被称为高阶元函数，是一种更为复杂和强大的元函数。

头文件<boost/mpl/apply_wrap.hpp>中定义了一系列的 apply_wrapN<>高阶元函数，它可以调用高阶元数据里的 apply<>元函数，声明如下：

```
template<typename F, typename arg1, typename arg2,... >
struct apply_wrapN
{
    typedef some_define type;
};
```

apply_wrap<>相当于在高级元函数调用时多了一个间接层，apply_wrapN<F,a1,a2,...>与 F::apply<a1,a2,...>等价。

我们也可以使用另一个高阶元函数 mpl::apply<>来达到同样的效果，写法更加简单，它没有 apply_wrap<>的数字后缀，缺省支持最多五个参数，但可以通过配置宏 BOOST_MPL_LIMIT_METAFUNCTION_ARITY 改变。

11.7.2 占位符

mpl 库在头文件<boost/mpl/placeholders.hpp>中定义了若干编译期占位符，它们都是高阶元数据，被用于构造 lambda 表达式，其定义与 boost.bind 中的占位符很相似：

```
typedef some_define  _;        //匿名占位符
typedef arg<1>       _1;       //占位符 1
typedef arg<2>       _2;       //占位符 2
...
typedef arg<n>       _n;       //占位符 n
```

这些编译期占位符实际上是参数元函数 arg<N>的别名，与运行时占位符的作用相似，可以选择参数列表中的第 n 个参数（默认最大为 5）。比较特殊的是匿名占位符“_”，它可以根据在表达式中的位置自动变为带数字的占位符，例如 bind<_,_>相当于 bind<_1,_2>。

示范占位符用法的代码如下：

```
#include <boost/mpl/placeholders.hpp>
#include <boost/mpl/apply_wrap.hpp>
#include <boost/mpl/apply.hpp>
using namespace boost::mpl;

int main()
{
  typedef apply_wrap2<_1, int, char>::type t1;        //获得第一个参数
  typedef apply<_3, int, char, float>::type t2;       //获得第三个参数

  assert((is_same<t1, int>::value));                  //验证占位符的效果
  assert((is_same<t2, float>::value));
}
```

11.7.3 bind 表达式

bind 表达式是一个高阶元函数，类似 boost.bind，它配合占位符实现参数的传递，可以在编译期绑定高阶元数据生成一个新的高阶元数据，就像 boost.bind 绑定函数对象那样。

bind 位于头文件<boost/mpl/bind.hpp>，声明如下：

```
template< typename F, typename a1, typename a2,... >
struct bind
{...};
```

示范 bind 用法的代码如下：

```
#include <boost/mpl/bind.hpp>
using namespace boost::mpl;

struct func1                              //一个简单的高阶元数据，类似函数对象
{
  template<typename T1>
  struct apply                            //apply<>元函数
  {   typedef T1 type;};                  //直接返回参数
};
int main()
{
  typedef bind<func1, _1> f1;             //用占位符绑定 func1 高阶元数据
```

```
                                                //也可写做 bind<func1, _>
    typedef apply<f1, int>::type data1;         //使用 apply<>调用
    assert((is_same<data1, int>::value));

    typedef apply<bind<func1, float>>::type data2; //直接绑定参数调用
    assert((is_same<data2, float>::value));
}
```

11.7.4 lambda 表达式

所谓“lambda 表达式”泛指含有占位符的表达式，例如 plus<_1,int_<10>>、add_const<_>。

注意：这些元函数在使用了占位符后就变成了普通的无参元函数（参数是占位符），无法完成元计算。它们也不是高阶元数据，也不能使用 apply_wrap<>元函数调用。例如，下面的代码无法通过编译：

```
typedef apply_wrap2<plus<_,_>,                  //调用 apply_wrap
    int_<1>, int_<2>>::type data0;              //编译失败
```

mpl 库在头文件<boost/mpl/lambda.hpp>提供了一个专门的元函数 lambda<>，它可以把一个占位符表达式包装成一个匿名高阶元数据，之后就可以被高阶元函数调用：

```
typedef apply_wrap2<lambda<plus<_,_>>::type,    //使用 lambda<>包装
    int_<1>, int_<2>>::type data0;
assert((data0::value == 3));
```

高阶元函数 apply<>比 apply_wrap<>更强大，它可以直接调用占位符表达式，这是因为它内部已经使用了 lambda<>来包装占位符表达式：

```
typedef apply<plus<_,_>,                        //注意，不需要 lambda<>包装
    int_<1>, int_<2>>::type data1;
assert((data1::value == 3));
```

lambda 表达式也可以应用于 bind，例如：

```
typedef bind<lambda<plus<_,_>>::type, _1,_2> f1;
typedef apply<f1, int_<1>, int_<2>>::type data2;
assert((data2::value == 3));
```

但这通常没有必要，因为 apply<>可以直接使用占位符表达式，比 bind 用起来更加灵

活方便。

11.7.5 算法的高级应用

有了高阶元数据、bind 和 lambda 表达式，算法的功能就更加强大了，它可以如同运行时标准算法一样，传入一个高阶元数据执行复杂的操作。

本节简要介绍这些使用 lambda 表达式算法的用法，不做深入的分析。

插入器

通用的插入器 inserter<S,Op>中的参数 Op 是一个 lambda 表达式，以一个初始状态 S 开始连续执行 Op 操作完成插入动作，故 back_inserter<C>相当于 inserter<C, push_back<_1,_2>>，front_inserter<C>相当于 inserter<s, push_front<_1,_2> >。

使用 inserter<>我们也可以做出与“插入”完全无关的操作，例如下面的代码执行了累加计算：

```
typedef range_c<int, 1, 11> rc;           //从 1 到 10 的整数区间
typedef mpl::copy<rc,                     //拷贝整数到插入器
  mpl::inserter<int_<0>, plus<_,_> >      //初值为 0，使用加法元函数
  >::type sum;
cout << sum::value;                       //输出元计算结果 55
```

查询算法

使用 lambda 表达式的查询算法如下，它们与标准算法很类似：

- find_if<C,P>　　：查找容器中满足谓词 P 的元素；
- count_if<C,P>　　：计算容器中满足谓词 P 的元素的个数；
- min_element<C,P=less<_,_>>　　：返回容器中的第一个最小元素的位置；
- max_element<C,P=less<_,_>>　　：返回容器中的第一个最大元素的位置；
- lower_bound<C,t,P=less<_,_>>：返回已经排序的容器中第一个可插入 t 的位置；
- upper_bound<C,t,P=less<_,_>>：返回已经排序的容器中最后一个可插入 t 的位置。

在使用这些查询算法时我们必须编写谓词高阶元数据定义元素的关系，例如，我们可以以类型的 sizeof 大小判定它们的顺序，示范代码如下：

```
#include <boost/mpl/vector.hpp>
#include <boost/mpl/lambda.hpp>
#include <boost/mpl/count_if.hpp>
#include <boost/mpl/find_if.hpp>
#include <boost/mpl/sizeof.hpp>
#include <boost/mpl/front.hpp>
#include <boost/mpl/min_element.hpp>
#include <boost/mpl/max_element.hpp>
using namespace boost::mpl;

int main()
{
  typedef mpl::vector<float, double,
      char, int, long> vec;               //容纳各种数值类型的容器

  assert((mpl::count_if<vec,              //count_if<>算法
      is_float<_> >::value==2));          //使用 is_float<>计算其中的浮点数
  assert((is_same<
          deref<mpl::find_if<vec,         //find_if<>查找 char 类型
          is_same<_, char>>::type>::type,
          char>::value));

  typedef lambda<                         //使用 sizeof_<>定义 lambda 表达式谓词
              less<sizeof_<_1>, sizeof_<_2>> > Comp;

  typedef deref<min_element<vec, Comp>::type>::type mint;//最小元素
  assert((is_same<mint, char>::value));

  typedef deref<max_element<vec, Comp>::type>::type maxt;//最大元素
  assert((is_same<maxt, double>::value));
}
```

变换算法

使用 lambda 表达式的部分变换算法如下，它们与标准算法同样很类似：

- copy_if<C,P> ：复制容器中满足谓词 P 的元素；

- replace_if<C,P,New> ：替换容器中满足谓词 P 的元素；
- remove_if<C,P> ：删除容器中满足谓词 P 的元素；
- unique<C,P> ：删除容器中连续的重复元素，是否连续由 P 确定；
- sort<C,P> ：对容器以谓词 P 作为排序准则排序；
- transform<C,Op> ：对容器内所有元素调用 Op 操作。

示范这些变换算法的代码如下：

```
#include <boost/mpl/vector_c.hpp>
#include <boost/mpl/copy_if.hpp>
#include <boost/mpl/replace_if.hpp>
#include <boost/mpl/remove_if.hpp>
#include <boost/mpl/sort.hpp>
#include <boost/mpl/unique.hpp>
using namespace boost::mpl;

int main()
{
  typedef vector_c<int, 5,3,7,2,6,4,2> vec;     //一个整数容器

  typedef copy_if<vec,                          //copy_if<>算法
      equal_to<modulus<_1, int_<2> > ,          //只复制偶数
              int_<0>> >::type vec2;            //注意元函数的组合使用

  typedef replace_if<vec2,                      //替换整数 6
      equal_to<_1, int_<6> >, int_<10>>::type vec3;

  typedef remove_if<vec3,                       //删除大于 5 的整数
      greater<_1, int_<5>>>::type vec4;

  typedef sort<vec4, less<_,_>>::type vec5;     //降序排序

  typedef unique<vec5, equal_to<_,_>>::type vec6;//删除重复元素
}
```

这段代码演示了大部分算法的用法，关于 transform<>的用法示例参见 11.8.2 小节。

11.8 mpl 的调试

模板元编程的调试是一项艰巨的工作，因为现有的大多数编译器对运行时调试支持的很好，但并没有对元编程提供特别的支持，我们无法像普通运行程序时一样设置断点、逐步跟踪并查看元数据的值，通常只能使用 typeid(T).name()来输出元计算中间结果检查，但这不总是可用的。

Boost 和 mpl 提供了一些简单的工具来方便元程序的调试，虽然仍不够完备，但已经可以部分减少我们的调试工作量。

11.8.1 断言

保证元程序正确性最基本的工具就是断言，boost.static_assert 库提供静态断言宏 BOOST_STATIC_ASSERT，它可以用在函数域、类域或名字空间域等程序的任何地方，相当于运行时的 assert 宏，例如：

```
typedef mpl::vector<int, char> vec;  //一个整数类型的容器
BOOST_STATIC_ASSERT((is_same<int,    //静态断言
  front<vec>::type>::value));        //断言前端元素是 int
```

静态断言宏 static_assert 比运行时断言 assert 好的地方是它执行在编译期，可以更早地检查出元程序的错误，而不必等到编译完成再运行。

但静态断言不是专门为元编程设计的，它虽然可以很好地保证元程序的正确性，但不能够给出有利于元编程诊断的更多可用信息，而且写法也较麻烦。mpl 库为此在头文件<boost/mpl/assert.hpp>特意定义了几个元编程断言，它们能够在元程序发生错误时提供有用的信息。

基本断言

宏 BOOST_MPL_ASSERT 是 mpl 中最常用的一个静态断言，它使用一个返回 bool_<>的值元函数 pred 作为参数，断定 pred::value 为真，调用形式是：

```
BOOST_MPL_ASSERT(( pred ));
```

注意宏的调用方式，必须使用两对圆括号，即使 pred 的模板参数列表中没有逗号。

BOOST_MPL_ASSERT 的用法与 BOOST_STATIC_ASSERT 类似，只是它不接受普通的

条件表达式，而只能是返回 bool_<>值元函数：

```
typedef mpl::vector<int, char> vec;  //一个整数类型的容器

BOOST_MPL_ASSERT((is_same<int,        //静态断言，两对圆括号
  front<vec>::type> ));               //注意不需要使用::value
BOOST_MPL_ASSERT((equal_to<           //验证容器的大小
  size<vec>::type, int_<3>>));        //发生一个断言错误
```

第二个断言会在编译时报出一个形如“************pred::***********”的错误，同时指出出错的行号，例如在 VC8 下为：

```
... ************boost::mpl::equal_to<N1,N2>::* *********** ...
```

否定断言

因为 BOOST_MPL_ASSERT 的测试条件是元函数，不能使用逻辑运算符，所以如果要验证否定条件就需要使用 not_<>元函数进行包装。为了简化否定条件的测试，mpl 定义了断言 BOOST_MPL_ASSERT_NOT。

BOOST_MPL_ASSERT_NOT 的用法与 BOOST_MPL_ASSERT 相同，也要求必须使用两对圆括号，只是判断的条件相反：

```
#include <boost/mpl/assert.hpp>
using namespace boost::mpl;

template<typename T>
struct my_operation                     //定义一个简单的元函数
{
  BOOST_MPL_ASSERT_NOT((is_pod<T>)); //要求是非 pod 类型
  BOOST_MPL_ASSERT((                    //要求有内部类型 value_type，并且是整型
         is_integral<typename T::value_type>));

  typedef typename next<T>::type type;//递增 mpl 整数类型

};

int main()
{
  typedef my_operation<int_<3>>::type t;//正确
  typedef my_operation<int>::type error;//编译错误
}
```

关系断言

仅有判断真或假的断言还是不够的，我们还需要判断逻辑关系的断言，同样的，为了解决使用元函数而带来的麻烦，mpl 提供了简化关系判断的宏，声明如下：

```
BOOST_MPL_ASSERT_RELATION(x, rel, y)
```

宏 BOOST_MPL_ASSERT_RELATION 的形式比较“怪异”，它有三个参数，x/y 是两个编译期整数（非包装器），rel 是一个合法的 C++关系操作符，如==、<。

BOOST_MPL_ASSERT_RELATION 不需要使用两对圆括号（使用了反而是错误的），示范代码如下：

```
BOOST_MPL_ASSERT_RELATION(int_<5>::value, >, 0);         //正确
OOST_MPL_ASSERT_RELATION(sizeof(int), <, sizeof(long)); //编译错误
```

编译的错误信息如下：

```
... ************assert_relation<...>::************ ...
```

定制消息的断言

运行时断言宏 assert 可以在断言同时用&&定制错误消息，例如：

```
assert(1>2 && "error message");
```

这种定制消息的功能对于程序的调试显然是很有用的，所以 mpl 也提供了一个类似功能的宏，声明如下：

```
BOOST_MPL_ASSERT_MSG( c, msg, types_ )
```

宏的三个参数的含义如下：

- c　　　：条件表达式，非元函数；
- msg　　：我们自行定制的消息，但它不是一个字符串，而是一个符合 C++语法的标志符，被宏用来生成一个仅用于编译报错的不完整类（struct）；
- types_ ：类型列表，用于定制输出断言失败时的类型信息，它可以是一个被圆括号包围的若干类型（可以为空），也可以是一个 types<...>结构。

示范 BOOST_MPL_ASSERT_MSG 用法的代码如下：

```
template<typename T>
struct my_operation                                    //对之前的元函数略做修改
{
  BOOST_MPL_ASSERT_MSG(!is_pod<T>::value,              //要求是非 pod 类型
      IS_POD_ERROR, (T));                              //定制错误消息和类型信息
};

int main()
{
  BOOST_MPL_ASSERT_MSG(1>2, DEMO_MESSAGE, ());         //简单的定制错误消息
  my_operation<int>::type;                             //编译错误
}
```

编译后输出的错误消息类型如下的形式：

```
************( main::DEMO_MESSAGE::* ***********
************( my_operation<T>::IS_POD_ERROR::* ***********
```

11.8.2 打印输出

在调试程序时打印日志是一个很重要的手段，对于目前的元编程来说，不能单步跟踪，那么就只有这么一个唯一的手段了。

在运行时查看元数据（类型）的信息很简单，只需要写 `cout << typeid(T).name()` 就可以了，但它的问题在于只能用在函数域，在名字空间域和类域无法发挥作用，而这些才是元编程大显身手的地方。

mpl 在头文件<boost/mpl/print.hpp>提供了一个 print<>元函数，它可以在编译期产生一个无关紧要的编译警告，以警告的形式打印输出类型信息，声明如下：

```
template <class T>
struct print: mpl::identity<T>                         //调用 identity<>元函数
{...};
```

print<>元函数用起来就像是标准 C 函数 printf()，我们不必使用 typedef 来定义它的返回值（因为不关心），只需要用::type 调用。

示范 print<>用法的代码如下：

```
#include <boost/mpl/print.hpp>                         //打印输出头文件
```

```
#include <boost/mpl/transform.hpp>
using namespace boost::mpl;

mpl::print<int>::type;                              //输出 int 类型
typedef mpl::vector<char, int_<5>, float> vec;    //类型容器
transform<vec, print<_>>::type;             //使用算法 transform<>打印容器里的类型

int main()
{}
```

这段代码无须运行，仅仅编译就可以了，会产生大量的编译错误信息，不过在这些繁杂的信息中我们可以很容易地用编辑器找到 print<>输出的类型信息，在 VC8 下是：

```
...boost::mpl::print<T>
with
[
    T=int
]
...boost::mpl::print<T>
with
[
    T=char
]
...boost::mpl::print<T>
with
[
    T=boost::mpl::int_<5>
]
...boost::mpl::print<T>
with
[
    T=float
]
```

11.9 mpl 实例研究

本节中我们将实际使用模板元编程技术和 mpl 实现一个动态加载 Windows 系统下 dll 接口函数的功能，这个例子来源于几年前笔者的一个实际项目，本书做了适当的简化。

大多数读者应该都对动态加载 dll 函数比较熟悉，基本的工作原理很简单，使用 WINAPI 函数 GetProcAddress() 获得函数指针然后再强制转型就可以了，我们来看在这里 mpl 能够发挥什么作用。

11.9.1　泛型编程版本

首先我们来看最简单的泛型版本，它很简单地封装了 Windows 的底层 API：

```
#include <exception>
#ifdef _MSC_VER                                 //仅支持 Windows 系统
#include "windows.h"
#else
#error please use windows
#endif

class DllManager                                //定义一个加载 dll 函数的包装类
{
public:
  DllManager(const char* szFileName)                //构造函数
  {
      m_hinstance = LoadLibrary(szFileName);        //使用文件名加载 dll
  }
  ~DllManager()                                     //析构函数
  {
      FreeLibrary(m_hinstance);                     //释放 dll
  }

  template<typename FuncType>                       //模板参数是函数指针类型
  FuncType load(const char* szFuncName)             //模板函数加载 dll 函数
  {
      FuncType pf = reinterpret_cast<FuncType>      //强制类型转换
          (GetProcAddress(m_hinstance, szFuncName)); //获得函数指针
      if (pf == 0)                                  //检查指针的正确性
      {
          throw std::exception("dll error");
      }
      return pf;
  }
private:
  HINSTANCE m_hinstance;                            //dll 句柄
};
```

类 DllManager 在构造时使用传入的文件名加载 dll，在析构时释放 dll。它的核心功能是模板函数 load()，模板参数是函数指针类型，调用 GetProcAddress()获得 dll 中导出函数的地址，然后使用转型操作符 reinterpret_cast<>转换为正确的函数指针类型。由于可能出现加载函数失败的情况，因此通常需要检查函数指针是否为空指针（为了代码清晰起见，接下来的代码都将忽略检查，请读者注意）。

假设我们在 testdll.dll 中有如下的两个导出函数：

```
int dll_func1(int x)                          //测试用导出函数 1
{ return x * x;}
int dll_func2(int x, int y)                   //测试用导出函数 2
{ return x + y;}
```

那么 DllManager 可以这样使用：

```
#include "DllManager.hpp"

typedef int (*Func1)(int);                    //首先定义函数指针
typedef int (*Func2)(int, int);

int main()
{
  DllManager dm("testdll.dll");               //加载 dll

  Func1 f1 = dm.load<Func1>("dll_func1");     //定义函数指针并加载
  Func2 f2 = dm.load<Func2>("dll_func2");     //定义函数指针并加载

  cout << f1(10) << endl;                     //调用加载的函数指针
  cout << f2(10, 20) << endl;
}
```

调用导出函数的代码也可以不使用函数指针变量暂存而直接调用：

```
cout << dm.load<Func1>("dll_func1")(10) << endl;
cout << dm.load<Func2>("dll_func2")(10, 20) << endl;
```

DllManager 使用了泛型编程，在一定程度上封装了原始的 Windows API，用起来也较原始手法要方便一些，但它仍然是运行时的，而且函数指针定义和函数名称定义之间缺乏紧固的联系，容易发生编写错误导致误用。

11.9.2 元编程第 1 版

本小节我们将使用 mpl 来改进 DllManager。

仔细分析加载 dll 这个例子我们可以把它分解为两部分：一部分是编译期的数据，包括文件名、函数名以及函数指针类型，另一部分是运行时的代码，包括加载 dll 文件和获取 dll 函数。泛型版本没有很好地把两者解耦，而是简单地在运行时混合在了一起，所以不够清晰。

因为我们现在拥有 mpl 这个强大的元编程工具，所以能够把编译期的数据分离出来。为了明确区分编译和运行时的数据，我们把程序实现为前端和后端两个部分：前端使用 mpl 定义编译期数据，后端使用前端数据实现运行时的功能。

前端

前端的数据有 dll 文件名、接口函数名和函数指针类型，前两者可以使用 mpl::string 表述，后者本身就是元数据，为了实现函数名与函数指针类型的对应关系我们还应该使用 mpl::map，最后要用一个 struct 把这些数据封装成为一个前端类。

前端类 dl_front 的实现代码如下：

```
#include <boost/mpl/vector.hpp>                          //元编程各种工具
#include <boost/mpl/string.hpp>
#include <boost/mpl/map.hpp>
#include <boost/mpl/at.hpp>
namespace mpl = boost::mpl;

struct dl_front                                          //前端类定义
{
  typedef mpl::string<'test','dll.','dll'> dll_name;//dll 文件名

  typedef int (*Func1)(int);                             //函数指针类型定义
  typedef int (*Func2)(int, int);

  typedef mpl::string<'dll_','func','1'> fun1_name;  //函数名定义
  typedef mpl::string<'dll_','func','2'> fun2_name;

  typedef mpl::map<                                      //函数名到函数指针类型的映射
      mpl::pair<fun1_name, Func1>,                       //使用 mpl::pair
      mpl::pair<fun2_name, Func2>
```

```
        >   map_fun;
};                                                   //前端类定义结束
```

dl_front 也可以看做是一个无参的非标准元函数，它可以返回与 dll 相关的多个相关数据，其中最重要的是 map_fun，它把函数名和函数指针类型紧密地联系在了一起。

后端

有了前端定义，后端 dl_back 的实现就容易多了，基本代码与 11.9.1 小节的 DllManager 很相似，只是操作的数据都变成了 dl_front 返回的元数据。

dl_back 我们实现为一个模板类，这样它就可以使用不同的前端类支持不同的 dll 加载功能（静态多态），而本身的代码保持稳定，构造函数和析构函数实现如下：

```
template<typename Front>            //模板参数是前端类
class dl_back
{
private:
  HINSTANCE m_hinstance;            //dll 句柄
public:
  dl_back()                         //构造函数，使用 c_str<>元函数获得 dll 文件名
  {
      m_hinstance = LoadLibrary(mpl::c_str<Front::dll_name>::value);
  }
  ~dl_back()                        //析构函数
  {
      FreeLibrary(m_hinstance);
  }
};
```

dl_back 的核心功能是成员模板函数 func<>()，它使用函数名作为索引，在前端的 map 中使用 at<>元函数查找对应的函数指针类型：

```
template<typename FuncName>
typename mpl::at<typename Front::map_fun, FuncName>::type//返回类型
func()
{
  typedef mpl::at<Front::map_fun, FuncName>::type
      result_type;                                      //typedef 以简化代码

  result_type pf = reinterpret_cast<result_type>
      (GetProcAddress(m_hinstance,mpl::c_str<FuncName>::value));
```

```
    return pf;
}
```

验证

前端和后端都已经实现，现在我们只需要传递一个在前端定义好的函数名元数据，后端就会使用模板元编程技术自动推导出对应的函数类型完成函数的加载，代码非常简洁。

```
dl_back<dl_front> dl;                                //使用前后端定义 dll 加载功能

cout << dl.func<dl_front::fun1_name>()(10) << endl;  //调用 dll 函数
cout << dl.func<dl_front::fun2_name>()(10,20) << endl;
```

注意，在调用成员函数 func<>()时我们除了要显式写出前端定义的函数名外，还必须使用两次 operator()，因为第一次 operator()调用只是获得了函数指针，第二次 operator()才是真正的 dll 接口函数调用。

11.9.3 元编程第 2 版

元编程的第一个版本应该说是比较成功的，它使用元编程技术分离了编译和运行两个时期的数据和代码，把 dll 相关的编译期数据都封装在了前端，而后端则使用了静态多态技术，任何符合前端定义的类都可以应用于后端，大大地增强了后端的稳定性和灵活性。

但第一版还有改进的余地。一方面我们可以在后端使用静态断言和元编程断言，约束前端的定义，可以避免前端的代码错误，另一方面我们可以增强成员函数 func<>()，直接传递参数减少一次 operator()的调用。断言的工作比较简单，读者可自行尝试完成，本小节实现第二个改进。

要减少一次 operator()的调用，这就要求函数 func<>()返回的是 dll 函数指针的返回类型而不是函数指针类型，同时传递相应数量的参数。前者可以使用 boost.result_of 结合 mpl 来自动推导函数的返回值类型[①]，后者可以使用模板参数数量重载来解决。

改进后的 func<>()代码如下：

```
#include <boost/utility/result_of.hpp>      //result_of 头文件
...                                          //前略
```

① 也可以使用 type_traits::function_traits 或者 function_types::result_type，读者可以自行尝试。

```
template<typename FuncName, typename T0>     //新增一个参数类型
typename boost::result_of<                   //使用 result_of 推导函数返回值
  typename mpl::at<typename Front::map_fun, FuncName>::type(T0)
  >::type
func(T0 t0)
{
  typedef mpl::at<Front::map_fun, FuncName>::type
      func_type;                             //从 mpl::map 得到函数指针类型

  func_type pf = reinterpret_cast<func_type> //获得函数指针
      (GetProcAddress(m_hinstance, mpl::c_str<FuncName>::value));

  return pf(t0);                             //直接调用函数指针
}
```

我们也可以直接使用第一版已经写好的 func<>() 函数直接返回函数指针来调用，这样可以相当程度上简化函数体内部的代码[①]：

```
template<typename FuncName, typename T0>
typename boost::result_of<
  typename mpl::at<typename Front::map_fun, FuncName>::type(T0)
  >::type
func(T0 t0)
{
  return func<FuncName>()(t0);    //直接调用获取函数指针的成员函数
}
```

第二个导出函数的重载形式也可以定义如下：

```
template<typename FuncName, typename T0, typename T1>    //多出两个参数
typename boost::result_of<
typename mpl::at<typename Front::map_fun, FuncName>::type(T0, T1)
>::type
func(T0 t0, T1 t1)
{
  return func<FuncName>()(t0, t1);    //直接调用获取函数指针的成员函数
}
```

① 使用之前编写的 func<>来简化代码有个小问题，就是无法实现无参版本的直接函数调用，这可以通过把 func<>改名（例如 funcPtr<>）来解决，这样直接调用版本就不会出现重载形式冲突了。

改进后的 func<>() 可以这样调用，写法更加简单：

```
cout << dl.func<dl_front::fun1_name>(10)<< endl;        //只有一对括号
cout << dl.func<dl_front::fun2_name>(10,20) << endl;
```

这个版本显然比第一版要更加方便好用，不过麻烦在于我们必须在后端写出多个参数的 func<>() 重载形式，如果有很多个不同参数数量的导出函数，我们必须写出大量的重复刻板的元编程代码[1]。这是模板元编程无法避免的缺陷（除非 C++提供可变模板参数的语言特性），但这些代码一旦写好就永远可用，这也是模板元编程的优点。

11.10 总结

本章我们讨论了 Boost 模板元编程库 mpl，它是进行模板元编程的主要工具。因为 mpl 内容庞杂，本章也只能阐述其中的部分内容。

mpl 基于现有的 C++标准创建了一套完整的元编程体系框架，使得我们无须从最基本的元编程概念开始工作，直接使用已经定义好的若干高级工具，极大地简化了元编程的开发工作。mpl 的体系结构完全是仿造 C++标准库的，许多概念都非常相似，所以只要理解了它在编译期运行的原理就能够较容易地掌握 mpl 的用法。

同 STL 一样，mpl 也有三大组成部分，分别是容器、迭代器和算法，此外还有类似函数对象的高阶元数据以及 bind 和 lambda 表达式。这些高级工具是 mpl 的核心，它们的实现很复杂，但用法却比较简单。

虽然 mpl 的高级元编程工具简化了开发工作，但现有的 C++编译器仍然缺乏调试元程序的能力，这给开发元程序带来了不小的麻烦。mpl 在 static_assert 静态断言外又提供了专用的元编程断言和类型打印功能，在很大程度上增强了元编程的正确性，提高了元编程的效率。

模板元编程目前仍然算是一个较新的编程范式，但已经得到了广泛的应用，尤其是在 Boost 库中，如 proto（领域嵌入式语言框架）、xpressive（静态正则表达式）、msm（元编程状态机）、fusion（混合了编译和运行时的代码）等都使用了大量的元编程技术和 mpl 组件，了解 mpl 对研究这些库会非常有帮助。而且，相信随着 C++的进一步发展，模板元编程最终将进入普通程序员的视野，本章的最后展示了一个使用 mpl 的小例子，也许有助于读者理解元编程的实际应用。

① 可以使用预处理元编程来简化多个类似模板函数的编写工作，这也是大多数元编程库的做法。

第12章

开发实践

Boost 是一个庞大而复杂的程序库，其广度和深度均超过了 C++98 中的标准库，但关于它的研究资料较标准库却是少得可怜，程序员往往会在学习的过程中不知不觉地陷入到 Boost 的代码海洋之中，虽然能够明白各个组件的功能和用法（并为之叹服），却不知道应该如何使用、在哪里使用，经常有种茫然四顾的感觉——就好像是手里拿了神兵利刃，但面对敌手却不懂得如何发力，徒生暴殄天物的遗憾。

本章中作者将结合自己使用 Boost 的经验，通过开发两个可用的 TCP 服务器程序实例来演示 Boost 的实际应用，希望能够给各位读者一些在真实项目中使用 Boost 的启发。需要说明的是，本章中的代码是一种“准产品代码”，虽然可以用于实际应用，但必须加以适当的调整，因为它不追求高效率高性能，主要目的还是示范、演示。同时为了方便叙述讲解，代码中没有使用名字空间限定，没有完整的异常/错误处理，测试也使用了最轻量级的测试工具（<boost/detail/lightweight_test.hpp>），读者需要注意。

12.1 基本工具

在任何一个工程项目中基本的工具模块都是必需的（也就是通常所说的 utils 模块），它们是整个项目的基础。Boost 程序库在这方面可以大有作为，因为它本身就已经提供了种类繁多琳琅满目的优秀工具类，涵盖几乎所有的程序开发领域，我们只需要在这些组件之上施加一层薄薄的封装（wrapper facard）以适应自己的需要即可。

本小节将利用 Boost 的一些组件实现几个最常用的工具，包括标准整数、并发线程池和日志，为 TCP 服务器开发做准备。

12.1.1　标准整数

虽然 C/C++提供了内置的整数类型定义，但不同的计算机硬件体系下整数的宽度（sizeof）是不同的：32 位硬件 int 宽度为 4 字节，而 64 位硬件 int 宽度为 8 字节，为了使我们的程序具有跨平台的兼容性，首先要定义程序中使用的整数类型，它是一切的基础。

遵循 C++标准库和 Boost 程序库的约定，这些整数类型都以后缀“_t”结尾，对于 Boost 中已经有定义的类型可以直接使用 using 关键字引入自己的名字空间：

```
// intdef.hpp [2011 Aug chrono]
#include <boost/cstdint.hpp>          //使用 Boost 的 integer 库

//精确宽度整数的定义
using boost::int8_t;                  //有符号 8 位整数
using boost::uint8_t;                 //无符号 8 位整数
using boost::int16_t;                 //有符号 16 位整数
using boost::uint16_t;                //无符号 16 位整数
using boost::int32_t;                 //有符号 32 位整数
using boost::uint32_t;                //无符号 32 位整数
using boost::int64_t;                 //有符号 64 位整数
using boost::uint64_t;                //无符号 64 位整数

//一些常用整数类型的定义
typedef boost::uint8_t  byte_t;       //字节类型
typedef boost::uint8_t  uchar_t;      //无符号字符类型
typedef unsigned short  ushort_t;     //无符号短整数
typedef unsigned int    uint_t;       //无符号整数
typedef unsigned long   ulong_t;      //无符号长整数

typedef boost::uint16_t word_t;       //word 整数
typedef boost::uint32_t dword_t;      //dword 整数
```

以上的代码中我们先定义了一些精确宽度的整数，然后使用这些标准整数类型定义了一些开发中常用的整数类型。这样，使用这些自定义的整数类型我们就成功地建立了一个中间层，屏蔽了因为可能的硬件差异而造成的整数宽度不一致的问题，使程序拥有了更好的可移植性。

对这些整数类型的简单单元测试代码如下：

```
void test_int()                       //使用 lightweight test 的单元测试函数
{
```

```
    BOOST_TEST_EQ(sizeof(int8_t), 1);         //int8_t 宽度为 1 字节
    BOOST_TEST_EQ(sizeof(uint8_t), 1);        //uint8_t 宽度为 1 字节

    BOOST_TEST_EQ(sizeof(int32_t), 4);        //int32_t 宽度为 4 字节
    BOOST_TEST_EQ(sizeof(int64_t), 8);        //int64_t 宽度为 8 字节

    BOOST_TEST_EQ(sizeof(ushort_t), 2);       //ushort_t 宽度为 2 字节
}
```

12.1.2　并发

boost.asio 是一个基于前摄器（proactor）模式的功能强大的并发库，它利用操作系统的 pool/epoll、kqueue、IO Overlap 等机制实现了无线程的并发操作，并且提供了 socket 通信、串口通信等功能，可用于开发高性能的网络应用程序。

本小节我们将结合 thread 库实现一个持有多个线程的 io_service_pool，其中的每一个线程里都运行着一个并发事件处理器(asio::io_service)，可以充分利用多核 CPU 的能力。

包含头文件

io_service_pool 需要包含的头文件如下：

```
// io_service_pool.hpp [2011 Aug chrono]
#include <algorithm>                                  //标准算法

#include <boost/assert.hpp>                           //基本工具
#include <boost/noncopyable.hpp>
#include <boost/foreach.hpp>
#include <boost/bind.hpp>
#include <boost/ref.hpp>
#include <boost/functional/factory.hpp>               //工厂函数对象
#include <boost/ptr_container/ptr_vector.hpp>         //指针容器

#define BOOST_ALL_NO_LIB
#include <boost/asio.hpp>                             //asio 并发库
#include <boost/thread.hpp>                           //thread 线程库
```

代码实现

因为 asio::io_service 是不可拷贝的，所以它最适合使用指针容器（第 5 章）来管

理，而多个线程的管理我们可以使用 boost::thread_group，比手工用容器存储 thread 指针或智能指针更方便好用：

```
class io_service_pool : boost::noncopyable
{
public:
  typedef boost::asio::io_service ios_type;              //io_service类型定义
  typedef boost::asio::io_service::work work_type;
  typedef boost::ptr_vector<ios_type> io_services_type;  //指针容器
  typedef boost::ptr_vector<work_type> works_type;

private:
  io_services_type m_io_services;          //指针容器存储io_service
  works_type m_works;                      //用work保证io_service运转

  boost::thread_group m_threads;           //线程池
  std::size_t m_next_io_service;           //用于轮询得到io_service
```

注意在 io_service_pool 的内部我们没有直接使用 asio::io_service、asio::io_service::work 等类型，而是使用 typedef 重新定义了若干个内部类型。这是一种在 STL、Boost 和现代 C++编程中的常用手法，也是个好习惯，相当于类型的 traits，不仅能够简化代码的编写，对于进一步的泛型编程也相当有益。

io_service_pool 的构造函数调用私有的 init()函数创建多个 io_service 对象并加入到指针容器中，代码如下所示：

```
public:
  explicit io_service_pool(int n = 4):     //构造函数
      m_next_io_service(0)
  {
      BOOST_ASSERT(n > 0);
      init(n);                             //初始化io_service指针容器
  }

private:
  void init(int n)                         //初始化io_service指针容器
  {
      for (int i = 0;i < n; ++i)
      {
          m_io_services.push_back(         //添加到指针容器
```

```
                boost::factory<ios_type*>()());     //使用 factory<>

            m_works.push_back(                       //添加到指针容器
                boost::factory<work_type*>()         //使用 factory<>
                    (m_io_services.back()));
        }
    }
```

成员函数 get() 使用简单的轮询算法(round-robin)调度多个线程中的 io_service (对于多数情况它足够用，读者可以根据自己的实际需要改用其他的算法):

```
public:
    ios_type& get() // round-robin 算法分配 io_service
    {
        if (m_next_io_service >= m_io_services.size())
        {   m_next_io_service = 0;  }

        return m_io_services[m_next_io_service++];//注意后置式++用法
    }
```

io_service_pool 使用 start() 和 run() 函数启动事件循环，前者是非阻塞而后者是阻塞的。start() 函数中我们使用 boost.foreach 算法遍历指针容器获得 io_service 对象，再使用线程执行 io_service::run() 启动事件循环；run() 函数则是在启动后调用线程组的 join_all() 阻塞等待。因为 io_service 是不可拷贝的，所以在传递给 bind 时必须使用 boost.ref 进行包装：

```
public:
    void start()                            //使用线程组启动 io_service
    {
        if (m_threads.size() > 0)           //已经启动了
        {   return; }

        BOOST_FOREACH(ios_type& ios, m_io_services)             //foreach 算法
        {
            m_threads.create_thread(        //创建线程启动 io_service 的事件循环
                boost::bind(&ios_type::run, boost::ref(ios))); //使用 ref
        }
    }

    void run()
```

```
    {
        start();                    //启动线程组里的 io_service
        m_threads.join_all();       //阻塞等待
    }
```

最后的 stop()同样是很简单，我们使用 std::for_each 算法来逐个调用 io_service 的成员函数 stop()，这里既可以使用 boost.bind 也可以使用 boost.mem_fn：

```
public:
    void stop()
    {
        m_works.clear();            //清除所有 io_service 的工作

        std::for_each(m_io_services.begin(), m_io_services.end(),
            boost::bind(&ios_type::stop, _1)); //使用 bind
                                    //也可以是 boost::mem_fn(&ios_type::stop)
    }
};//io_service_pool 定义结束
```

12.1.3 日志

虽然 Boost 社区已经在 2010 年中接纳了 boost.log 成为 Boost 程序库的一部分，但很遗憾在一年之后的今天它还没有正式发布，在此我们只能使用一个替代产品：log4cplus。

log4cplus 是一个仿造著名的 log4j 的日志库，其用法与 log4j 非常相似，支持多线程，目前的最新版本是 1.04。log4cplus 具体的编译和配置过程本书不做详细介绍（自带的文档已经写的很清楚了，并且体贴地提供了多个版本的 VC 工程，Unix 下就更简单了），这里只对它做一个简单的封装来方便使用。

头文件 log_init.h 声明了一个初始化函数和一个简单的日志宏：

```
//log_init.h [2011 Aug chrono]
#define LOG4CPLUS_STATIC            //静态库方式使用 log4cplus
#include <log4cplus/logger.h>

void log_init();
#define LOG_TRACE(x) LOG4CPLUS_TRACE(log4cplus::Logger::getRoot(),x)
```

日志的实现我们采用配置文件的方式，这样可以很方便地变动日志的输出形式：

```
//log_init.cpp [2011 Aug chrono]
#include "log_init.h"

#include <log4cplus/configurator.h>
using namespace log4cplus;

#define LOG4PLUS_CONFIG_FILE "log4cplus.cfg" //配置文件名

#ifdef _MSC_VER
#pragma comment(lib, "ws2_32.lib")          //windows 下需链接 winsock 库
#endif

void log_init()                             //调用 log4cplus 的配置器完成配置
{
  PropertyConfigurator::doConfigure(LOG4PLUS_CONFIG_FILE);
}
```

本章中我们使用一个最简单的配置，直接向屏幕输出，信息包含本地时间和线程号：

```
log4cplus.rootLogger=TRACE,ConsoleAppender
log4cplus.appender.ConsoleAppender=log4cplus::ConsoleAppender
log4cplus.appender.ConsoleAppender.layout=log4cplus::PatternLayout
log4cplus.appender.ConsoleAppender.layout.ConversionPattern={%D}{%4t}-%m %n
```

实际项目中可以把日志输出到一个转储文件中，例如：

```
log4cplus.rootLogger=TRACE,FileAppender

log4cplus.appender.FileAppender=log4cplus::RollingFileAppender
log4cplus.appender.FileAppender.File=testlog.log
log4cplus.appender.FileAppender.layout=log4cplus::PatternLayout
log4cplus.appender.FileAppender.layout.ConversionPattern={%D}{%-5p}{%4t} -
%m %n
```

日志类的用法可参见 12.2.6 小节。

12.2 第一个 TCP 服务器

本节我们将使用 asio 库开发一个异步的 TCP 服务器，它的结构比较简单，基本不使用

多线程，在此先介绍一下程序中的各个类：

- tcp_buffer ：简单封装了 asio::streambuf，是一个读写缓冲区；
- tcp_server ：使用了 io_service_pool，监听端口创建 TCP 连接；
- tcp_session：实现 TCP 连接的各种功能。

这三个类的 UML 描述如图 12-1 所示：

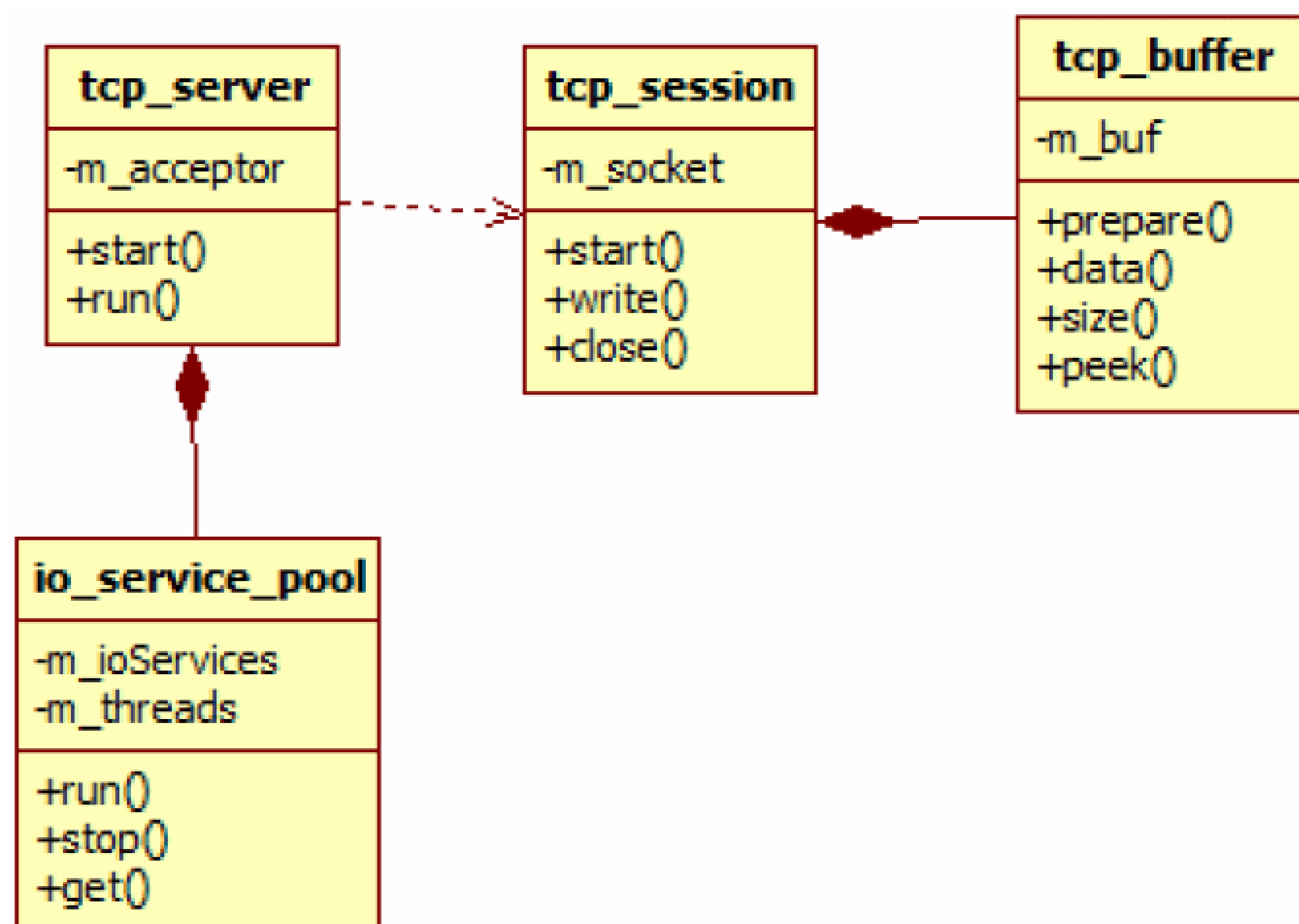


图 12-1 TCP 服务器类图

12.2.1 tcp_buffer

tcp_buffer 是一个非常简单的类，它封装了 asio::streambuf，用于 TCP 通信时的数据读写。asio::streambuf 派生自 std::basic_streambuf，它拥有两个独立的输入输出字符序列，是一个双向流（bidirectional），详细的信息请读者参考 Boost 资料。[①]

tcp_buffer 的接口基本上是模仿了 asio::streambuf，但做了一些适当的简化，方便使用，需包含的头文件如下：

```
// tcp_buffer.hpp [2011 Aug chrono]
#define BOOST_ALL_NO_LIB
#include <boost/asio.hpp>              //asio 库
#include <boost/config/suffix.hpp>     //静态成员变量
#include <boost/cast.hpp>              //数字转换
```

① 实际上 asio::streambuf 内部持有一个 vector，用几个读写指针实现了缓冲区的操作。

首先仍然是内部类型定义（traits）和成员变量：

```
class tcp_buffer                                    // tcp_buffer 类
{
public:
  typedef std::size_t size_type;                    //内部类型定义
  typedef boost::asio::streambuf streambuf_type;
  typedef streambuf_type::const_buffers_type const_buffers_type;
  typedef streambuf_type::mutable_buffers_type mutable_buffers_type;

private:
  BOOST_STATIC_CONSTANT(size_type, BUF_SIZE = 512); //定义缓冲区缺省大小
  streambuf_type m_buf;                             //asio 流缓冲
```

tcp_buffer 的接口很小，有四个用于接收数据。prepare()准备一块固定大小的输出缓冲区用于接收数据，retrieve()把收到的数据拷贝到流的输入序列，然后就可以用 size()和 peek()访问输入缓冲：

```
public:
  mutable_buffers_type prepare(size_type n = BUF_SIZE)   //用于接收数据
  {   return m_buf.prepare(n);}

  void retrieve(size_type n)                             //获取接收到的 n 个字节
  {   m_buf.commit(n);}

  size_type size() const                                 //获得可读的字节数量
  {   return m_buf.size();}

  const char* peek() const                               //查看可读的字节
  {   return boost::asio::buffer_cast<const char*>(m_buf.data());}
```

发送数据相对简单，tcp_buffer 提供两个接口。append()把 n 个字节的数据写入缓冲区，data()返回一个可传递给 asio 写函数的 const_buffers_type 类型：

```
public:
  void append(const void* data, size_type len)           //写数据到缓冲区
  {
      m_buf.sputn(static_cast<const char*>(data),
          boost::numeric_cast<std::streamsize>(len));    //使用 numeric_cast
  }
```

```
  const_buffers_type data() const             //用于发送数据
  {   return m_buf.data();}
```

最后是成员函数 consume()，在收发完数据后我们必须调用它，告诉缓冲区已经消费了多少个字节：

```
public:
  void consume(size_type n)                   //指示缓冲区消费 n 个字节
  {   m_buf.consume(n);}
};        //tcp_buffer 定义结束
```

tcp_buffer 的用法见 12.2.3 小节。

12.2.2 tcp_server

tcp_server 是 TCP 服务器的“引擎”，但它非常的简单，主要工作就是使用 asio::ip::tcp::acceptor 监听端口，一旦有连接事件发生就创建一个 tcp_session，之后的通信工作完全由 tcp_session 并发处理——tcp_server 自己只负责监听。

头文件

tcp_server.h 需包含如下的头文件：

```
// tcp_server.h [2011 Aug chrono]
#include "intdef.h"                           //整数类型定义
#include "tcp_session.h"                      //tcp_session 的定义，见 12.2.3 小节
#include "io_service_pool.hpp"                //并发线程池
```

tcp_server 的类声明代码如下：

```
class tcp_server                              // tcp_server 类
{
public:
  typedef io_service_pool::ios_type ios_type;//内部类型定义，使用 traits

  tcp_server(ushort_t port, int n = 1);       //构造函数
  void start();                               //非阻塞启动
  void run();                                 //阻塞启动

private:
  io_service_pool m_ios_pool;                 //并发线程池
```

```
    typedef boost::asio::ip::tcp::acceptor acceptor_type;//接收器类型定义
    acceptor_type m_acceptor;            //接收器

    void start_accept();                 //启动端口监听，异步接受连接
    void handle_accept(const boost::system::error_code& error,
        tcp_session_ptr session);        //accept 的异步处理函数
};      //tcp_server 声明结束
```

tcp_server 中最重要的成员就是 m_acceptor，它是接收器-连接器模式中的接收器，专门负责异步监听端口，当有连接事件发生时调用处理函数 handle_accept()。

handle_accept()函数参数中的 tcp_session_ptr 是 tcp_session 的智能指针，定义是：

```
typedef boost::shared_ptr<tcp_session> tcp_session_ptr;
```

使用 shared_ptr 来代替原始指针可以让我们更安全地在并发环境中使用，不必担心有内存泄漏。

实现文件

tcp_server.cpp 需包含如下的头文件，并打开 boost::asio 名字空间。

```
// tcp_server.cpp [2011 Aug chrono]
#include "tcp_server.h"

#include <boost/bind.hpp>
#include <boost/functional/factory.hpp>          //使用工厂函数对象
using namespace boost;
using namespace boost::asio;
```

tcp_server 的构造函数初始化 io_service_pool 和 acceptor，并调用 start_accept()开始异步监听端口：

```
tcp_server::tcp_server( ushort_t port,  int n ):
  m_ios_pool(n),                                 //并发线程池
  m_acceptor(m_ios_pool.get(),                   //获得一个 io_service
      ip::tcp::endpoint(ip::tcp::v4(), port))    //设置监听端口
{
  start_accept();                                //开始监听端口
}
```

start_accept()创建一个 tcp_session 指针，然后启动 acceptor 的异步接受连

接，用 bind 绑定了接受事件发生时的处理函数：

```
void tcp_server::start_accept()
{
  tcp_session_ptr session =
      factory<tcp_session_ptr>()(m_ios_pool.get());  //创建 tcp_session

  m_acceptor.async_accept( session->socket(),        //启动异步接受
      bind(&tcp_server::handle_accept, this,         //绑定处理函数
              placeholders::error, session));        //使用 asio 的占位符
}
```

async_accept()函数要求处理函数必须是 void handler(error_code& error)的形式，所以我们必须用 bind 来把函数适配成它要求的形式。handle_accept()是接收事件的处理函数，它启动 TCP 连接：

```
void tcp_server::handle_accept( const system::error_code& error,
                                tcp_session_ptr session )
{
  start_accept();          //启动一个新的异步接受操作

  if (error)               //错误处理
  {
      session->close();    //关闭连接
      return;
  }

  session->start();        //启动 TCP 连接
}
```

handle_accept()的工作很简单。首先它必须立即调用 start_accept()再启动异步接收操作，否则当函数执行完毕或出错后 m_acceptor 将因为异步事件完成而无法再接受连接。如果接受连接时有错误发生（error!=0）它就简单地关闭连接，如果连接正常，那么它就调用 tcp_session 的 start()启动 TCP 连接，开始数据的收发操作。

handle_accept()的最后一行代码也可以写成如下的形式：

```
session->io_service().dispatch(
    bind(&tcp_session::start, session));
```

这将把 tcp_session 的 start()的调用交给 io_service，由 io_service 来决定

何时执行这个函数，并发度会更高一些。[①]

tcp_server 的最后两个 run() 和 start() 非常简单，仅仅是转发给 io_service_pool：

```
void tcp_server::start()
{ m_ios_pool.start();}                    //非阻塞

void tcp_server::run()
{ m_ios_pool.run();}                      //阻塞
```

关于 io_service_pool

上面的代码中 tcp_server 与 io_service_pool 是强联系的组合关系，我们也可以把它们的联系放松一些，变为稍弱的聚合关系，把持有 io_service_pool 的实例改为引用，即：

```
private:
  io_service_pool& m_ios_pool;            //不是对象实例，而是引用
```

tcp_server 的构造函数也应该对应修改[②]：

```
tcp_server::tcp_server( ushort_t port,  int n ):
  //m_ios_pool(n),                        //不使用实例变量
  m_ios_pool(*factory<io_service_pool*>()(n)),//在堆上新建一个对象
  m_acceptor(m_ios_pool.get(),
      ip::tcp::endpoint(ip::tcp::v4(), port))
{
  start_accept();                         //其他代码不变
}
```

可以再为 tcp_server 增加一个构造函数，它接受一个外部的 io_service_pool 实例：

```
tcp_server::tcp_server( io_service_pool& ios, ushort_t port ):
  m_ios_pool(ios),                        //使用外部的实例变量
  m_acceptor(m_ios_pool.get(),
```

① io_service 提供两个异步调用方法：dispatch() 和 post()，两者功能相同，但回调的执行时机不同，dispatch() 的回调可能立即执行，而 post() 则必定在本次调用之后执行。

② 这里我们使用 factory<>直接在堆上创建了一个 io_service_pool 对象，不能使用 scoped_ptr 或者 shared_ptr，因为在离开作用域后智能指针会自动销毁，使 m_ios_pool 变成一个无效的引用。如果要确保没有内存泄漏可以把 m_ios_pool 由引用变成一个智能指针。

```
        ip::tcp::endpoint(ip::tcp::v4(), port))
  {
    start_accept();          //其他代码不变
  }
```

这样 tcp_server 与 io_service_pool 两者就可以彼此独立了，一个 io_service_pool 可以为多个 tcp_server 提供并发服务，一个 tcp_server 也可以连接不同的 io_service_pool。

12.2.3 tcp_session

tcp_session 是整个 TCP 服务器程序中最重要的类，负责处理网络通信，实现了数据收发的核心功能。

头文件

tcp_session.h 需包含如下的头文件：

```
// tcp_session.h [2011 Aug chrono]
#include "tcp_buffer.hpp"                       //缓冲类

#include <boost/smart_ptr.hpp>                  //智能指针
#include <boost/enable_shared_from_this.hpp> //从 this 创建 shared_ptr

#define BOOST_ALL_NO_LIB
#include <boost/asio.hpp>                       //asio 库
```

tcp_session 使用 boost::enable_shared_from_this，这样它就可以使用 shared_ptr 管理自己：

```
class tcp_session:
  public boost::enable_shared_from_this<tcp_session>
{
public:
  typedef boost::asio::ip::tcp::socket socket_type;  //内部类型定义
  typedef boost::asio::io_service ios_type;
  typedef tcp_buffer buffer_type;

private:                                              //成员变量
  socket_type m_socket;                               //asio 的 socket 封装
```

```
    buffer_type m_read_buf;     //读缓冲
    buffer_type m_write_buf;    //写缓冲
```

成员变量 m_socket 是 tcp_session 的核心，它是 asio 库提供的 tcp 通信主要功能类。

tcp_session 的成员函数较多，可以分为两类。一类是成员访问，用于获得内部成员的引用，另一类是 TCP 通信功能：

```
public:
    tcp_session(ios_type& ios);      //构造函数

    socket_type& socket();           //获得 socket
    ios_type& io_service();          //获得 io_service
    buffer_type& read_buf() ;        //获得读缓冲
    buffer_type& write_buf() ;       //获得写缓冲

    void start();                    //启动 TCP 连接，开始读数据
    void close();                    //关闭 TCP 连接

    void write();                    //异步发送写缓冲里的数据
    void write(const void* data, std::size_t len); //发送指定的数据

private:
    void read();                                        //异步读数据到读缓冲

    void handle_read(const boost::system::error_code& error,
        size_t bytes_transferred);                      //读处理函数
    void handle_write(const boost::system::error_code& error,
        size_t bytes_transferred);                      //写处理函数
};                                                      // tcp_session 定义结束

typedef boost::shared_ptr<tcp_session> tcp_session_ptr;//共享指针定义
```

构造函数和成员访问函数

为了便于调试，tcp_session.cpp 中我们使用了标准流，用来输出调试信息（日志的使用见后）：

```
// tcp_session.cpp [2011 Aug chrono]
#include "tcp_session.h"

#include <iostream>                                  //标准流库
#include <string>
using namespace std;                                 //打开 std 名字空间

#include <boost/bind.hpp>
using namespace boost;
using namespace boost::asio;
```

tcp_session 的构造函数和成员访问函数均非常简单，都只有一行代码：

```
tcp_session::tcp_session( ios_type& ios ):           //构造函数
  m_socket(ios)   {}

tcp_session::socket_type& tcp_session::socket()      //获得 socket
{ return m_socket;}

tcp_session::ios_type& tcp_session::io_service()     //获得 io_service
{ return m_socket.get_io_service();}

tcp_session::buffer_type& tcp_session::read_buf()    //获得读缓冲
{ return m_read_buf;}

tcp_session::buffer_type& tcp_session::write_buf()  //获得写缓冲
{ return m_write_buf;}
```

TCP 连接的打开和关闭

TCP 连接的打开和关闭功能比较简单。在启动 TCP 连接后输出一条简单的提示信息，然后调用 read()启动异步读操作；关闭 TCP 连接时先停止收发操作，然后关闭 socket：

```
void tcp_session::start()               //启动 TCP 连接，开始读数据
{
  cout << "session start" << endl;     //输出一条简单的提示信息

  read();                               //调用 read()启动异步读操作
}

void tcp_session::close()               //关闭 TCP 连接
```

```
{
  boost::system::error_code ignored_ec;  //一个错误代码，不使用
  m_socket.shutdown(ip::tcp::socket::shutdown_both, ignored_ec);
  m_socket.close(ignored_ec);
}
```

TCP 读操作

tcp_session 的读操作分为两步，先用 read()启动异步读，设置处理函数，然后在 handle_read()里处理接收到的数据，再调用 read()启动异步读：

```
void tcp_session::read()                                  //启动异步读
{
  m_socket.async_read_some(                               //asio 的异步读
      m_read_buf.prepare(),                               //使用读缓冲
      bind(&tcp_session::handle_read, shared_from_this(),  //绑定函数
          placeholders::error, placeholders::bytes_transferred)
      );
}

void tcp_session::handle_read( const system::error_code& error,
                              size_t bytes_transferred )
{
  if (error)                                          //错误处理
  {
      close();
      return;
  }

  cout << "read size" << bytes_transferred << endl;   //调试信息

  m_read_buf.retrieve(bytes_transferred);             //缓冲区接收数据
  cout << string(m_read_buf.peek(), bytes_transferred) << endl;

  write(m_read_buf.peek(), bytes_transferred);        //原样发送回客户端

  m_read_buf.consume(bytes_transferred);              //n 个字节已经被消费
  read();                                             //再启动一个异步读
}
```

在函数 read()中启动异步读设置处理函数时我们必须使用 shared_from_this()而

不能直接使用 this，因为原始指针不能够保证在并发环境下的安全性，有可能会发生一些莫名其妙的错误。

handle_read()中我们直接把收到的数据原样发送回客户端，相当于实现了 echo 协议（rfc 862）。

TCP 写操作

tcp_session 的写操作分为三步，先把数据添加到写缓冲，然后用 write()启动异步写，设置处理函数，最后在 handle_ write()里编写发送完成后的代码：

```
void tcp_session::write( const void* data, size_t len )
{
  cout << "write:" << len << endl;    //调试信息
  cout << static_cast<const char*>(data) << endl;

  m_write_buf.append(data, len);      //把数据添加到写缓冲

  write();                            //启动异步写
}

void tcp_session::write()             //发送写缓冲里的数据
{
  m_socket.async_write_some(          //异步写
      m_write_buf.data(),             //使用写缓冲
      bind(&tcp_session::handle_write, shared_from_this(),
          placeholders::error, placeholders::bytes_transferred)
      );
}

void tcp_session::handle_write( const system::error_code& error,
                                size_t bytes_transferred )
{
  if (error)                          //错误处理
  {
      close();
      return;
  }

  m_write_buf.consume(bytes_transferred);//n 个字节已经被消费
```

```
    cout << "write complete" << endl;                    //调试信息
}
```

12.2.4 验证

现在 TCP 服务器的代码就开发完成了，我们需要写一个 main() 函数把服务器启动起来，代码非常简单：

```
#include "tcp_server.h"                                  //tcp 服务器

int main()
{
    cout << "server start" << endl;                      //调试信息

    tcp_server svr(6677);                                //使用 6677 端口, 一个线程
    svr.run();                                           //启动服务器
}
```

简单的验证用客户端代码如下：

```
#define BOOST_ALL_NO_LIB
#include <boost/asio.hpp>
using namespace boost;
using namespace boost::asio;

int main()
{
    cout << "client start" << endl;

    io_service ios;
    ip::tcp::socket sock(ios);

    ip::tcp::endpoint ep(
        ip::address::from_string("127.0.0.1"), 6677);  //本地地址
    sock.connect(ep);                                    //连接到 echo 服务器

    string str("hello world");
    sock.write_some(buffer(str));                        //发送数据

    vector<char> v(100, 0);
```

```
    size_t n = sock.read_some(buffer(v));           //接收数据
    cout << &v[0] << endl;                          //显示接收到的数据
};
```

12.2.5　使用回调函数

我们编写的 TCP 服务器功能基本完备，初步可用，但 tcp_session 有一个缺点，它把数据的收发功能与处理功能紧耦合在了一起，不能够很好地变动或者扩展功能，缺乏足够的灵活性。

为了去除这种耦合性，可以使用回调函数的方式，给 tcp_session 添加一些回调函数，在建立/关闭连接、读/写数据时执行一些特定的处理操作，这些操作是与 tcp_session 的数据收发完全无关的（可以把这些回调函数看作是运行在 server 上的 service）。

回调函数接口

我们定义一个“智能接口类”tcp_handler 来封装回调，它类似于使用虚函数的纯接口类，但它不用语言级别的纯虚函数，而是用 boost::function 这个“智能函数指针”定义了四个回调函数：

```
#include <boost/smart_ptr.hpp>
#include <boost/function.hpp>

class tcp_session;                                  //前置声明
struct tcp_handler                                  //回调函数类，“智能接口”
{
    typedef const boost::shared_ptr<tcp_session>& tcp_session_type;

    //处理打开关闭连接
    boost::function<void(tcp_session_type)> handle_open;
    boost::function<void(tcp_session_type)> handle_close;

    //处理读写数据
    boost::function<void(tcp_session_type, std::size_t)> handle_read;
    boost::function<void(tcp_session_type, std::size_t)> handle_write;
};                                                  //tcp_handler 定义结束
```

同真正的接口一样，特定的回调函数操作应该派生自 tcp_handler，编写具体的回调处理函数，并在构造函数中赋值给 function 对象[①]。

① boost::function 通常要搭配 boost::bind 使用，可以把任意的可调用物转化为一个函数对象（闭包）。

tcp_session 的修改

因为使用了回调函数，所以 tcp_session 的代码必须做出修改，去掉原代码中处理数据的代码，而改为调用回调函数。

tcp_handler 必须传递给 tcp_session，这里我们选择修改 start() 函数而不是构造函数（其实两者的效果无太大差别，但在 start() 里会更灵活一些，可以中途变动 handler）：

```
class tcp_session:
  public boost::enable_shared_from_this<tcp_session>
{
public:
  void start(tcp_handler handler = tcp_handler());   //修改接口参数
private:
  tcp_handler m_handler;                              //保存回调函数的成员变量
  ...                                                 //其他代码不变
};
```

其余的代码修改如下：

```
void tcp_session::start(tcp_handler handler) //启动 TCP 连接
{
  m_handler = handler;                       //保存回调函数
  if (m_handler.handle_open)                 //如果回调函数不空则回调
  {   m_handler.handle_open(shared_from_this());}

  read();                                    //开始异步读
}
void tcp_session::close()                    //关闭 TCP 连接
{
  boost::system::error_code ignored_ec;
  m_socket.shutdown(ip::tcp::socket::shutdown_both, ignored_ec);
  m_socket.close(ignored_ec);

  if (m_handler.handle_close)                //如果回调函数不空则回调
  {   m_handler.handle_close(shared_from_this());}
}
void tcp_session::handle_read( const system::error_code& error,
                               size_t bytes_transferred )
```

```
{
  if (error)                                   //错误处理
  {
      close();
      return;
  }

  m_read_buf.retrieve(bytes_transferred);      //接收数据

  if (m_handler.handle_read)                   //如果回调函数不空则回调
  {   m_handler.handle_read(shared_from_this(), bytes_transferred);}

  m_read_buf.consume(bytes_transferred);       //消费 n 个字节
  read();                                      //再启动异步读
}
void tcp_session::handle_write( const system::error_code& error,
                                size_t bytes_transferred )
{
  if (error)
  {
      close();
      return;
  }

  m_write_buf.consume(bytes_transferred);      //消费 n 个字节

  if (m_handler.handle_write)                  //如果回调函数不空则回调
  {   m_handler.handle_write(shared_from_this(), bytes_transferred);}
}
```

读者需要注意函数 tcp_session::handle_read()，我们先执行了回调函数然后自动消费缓冲区里的字节，这个操作也可以要求回调函数手动完成，把数据的使用权完全交给用户。

tcp_server 的修改

tcp_server 的变动很小，只需要在启动 tcp_session 的时候添加 handler 类即可：

```
void tcp_server::handle_accept( ...)
{
  ...                                          //代码同前
```

```
    session->start(tcp_handler());          //传入回调函数
}
```

更进一步地，我们可以把 tcp_server 改为模板类，把 handler 类作为模板参数传递给 tcp_server（策略模式），这样的写法更加简单优雅。

模板类形式的 tcp_server 可以如下使用：

```
template<typename Handler>                  //用模板参数传递 handler 类
class tcp_server
{
    ...                                     //代码同前
private:
    typedef Handler handler_type;           //类型定义
    handler_type m_handler;                 //类内部的成员变量保存 handler

public:
    void tcp_server::handle_accept(...)
    {
    ...                                     //代码同前
        session->start(m_handler);          //传递 handler
    }
};                                          //tcp_server 定义结束
```

这个模板类 tcp_server 有一个小缺陷，它要求回调函数接口 Handler 必须是可缺省构造的，否则就无法使用。如果 Handler 需要参数才能构造，可以考虑修改 tcp_server 的构造函数，传递参数完成 Handler 的初始化。

12.2.6 简单协议的实现

本小节我们将使用回调函数来实现几个简单的 TCP 协议，这些协议包括 echo（rfc862）、discard（rfc863）、datetime（rfc867）、time（rfc868）、chargen（rfc864）。

echo 协议

之前在 tcp_session 中我们已经初步实现了 echo 协议，它把收到数据原样发送回客户端，所以 echo_handler 的代码只需要从原代码中提取出来即可：

```
#include "tcp_handler.h"
#include <boost/bind.hpp>                  //使用 bind 配合 function

class echo_handler : public tcp_handler   //继承 tcp_handler
{
public:
  echo_handler()                          //构造函数，赋值 function 对象，使用 bind
  {
      handle_open = boost::bind(          //处理打开
          &echo_handler::echo_handle_open, this, _1);
      handle_close = boost::bind(         //处理关闭
          &echo_handler::echo_handle_close, this, _1);
      handle_read = boost::bind(          //处理读
          &echo_handler::echo_handle_read,this, _1, _2);
      handle_write = boost::bind(         //处理写
          &echo_handler::echo_handle_write,this, _1, _2);
  }
```

四个具体回调函数（注意是私有的）实现如下，其中使用了 12.1.3 小节定义的日志功能：

```
private:
  void echo_handle_open(tcp_session_type session)
  {
      LOG_TRACE("session start from:");  //输出客户端的 ip 地址
      LOG_TRACE(session->socket().remote_endpoint().address());
  }
  void echo_handle_close(tcp_session_type session)
  {
      LOG_TRACE("session close");         //简单地显示消息
  }
  void echo_handle_read(tcp_session_type session, std::size_t n)
  {
      LOG_TRACE("read size:" << n);       //显示收到的数据
      LOG_TRACE(string(session->read_buf().peek(), n));

      session->write(session->read_buf().peek(), n);//回写
  }
  void echo_handle_write(tcp_session_type session, std::size_t n)
  {
      LOG_TRACE("write complete:" << n); //显示发送的字节数
  }
};                                        // echo_handler 定义结束
```

discard 协议

discard 协议抛弃接收的字节，而 tcp_handler 默认就是不处理数据，所以我们只需要在 handle_read 接口输出一些调试信息就可以了：

```
class discard_handler : public tcp_handler   //继承 tcp_handler
{
public:
  discard_handler()
  {
      handle_open = boost::bind(              //处理打开
          &discard_handler::discard_handle_open, this, _1);
      handle_read = boost::bind(              //处理读
          &discard_handler::discard_handle_read,this, _1, _2);
  }
private:
  void discard_handle_open(tcp_session_type session)
  {
      LOG_TRACE("discard start from:");       //调试信息
      LOG_TRACE(session->socket().remote_endpoint().address());
  }
  void discard_handle_read(tcp_session_type session, std::size_t n)
  {
      LOG_TRACE("read size:" << n);           //调试信息
      LOG_TRACE(string(session->read_buf().peek(), n));
  }
};// discard_handler 定义结束
```

daytime 协议

daytime 协议在连接时向客户端返回当前的时间，所以我们需要实现 handle_open 接口，并在发送完毕的 handle_write 中关闭通信：

```
#include <boost/typeof/typeof.hpp>                    //BOOST_AUTO
#include <boost/date_time/posix_time/posix_time.hpp> //日期时间库

class daytime_handler : public tcp_handler            //继承 tcp_handler
{
public:
  daytime_handler()
```

```
    {
        handle_open = boost::bind(                          //处理打开
            &daytime_handler::daytime_handle_open, this, _1);
        handle_write = boost::bind(                         //处理写
            &daytime_handler::daytime_handle_write,this, _1, _2);
    }
private:
    void daytime_handle_open(tcp_session_type session)
    {
        LOG_TRACE("daytime start from:");                   //调试信息
        LOG_TRACE(session->socket().remote_endpoint().address());

        using namespace boost::posix_time;                  //使用 date_time 库
        BOOST_AUTO(ptime, microsec_clock::local_time());    //当前时间
        std::string str = to_simple_string(ptime);          //转换为字符串

        session->write(str.c_str(), str.size());            //发送时间字符串
    }
    void daytime_handle_write(tcp_session_type session, std::size_t n)
    {
        LOG_TRACE("write complete:" << n);                  //完成发送
        session->close();                                   //关闭 TCP 通信
    }
};// daytime _handler 定义结束
```

time 协议

time 协议与 daytime 类似，只是它返回给客户端是一个自 epoch（1970-01-01 00:00:00）以来的秒数，实现与 daytime 几乎一样：

```
class time_handler : public tcp_handler                     //继承 tcp_handler
{
public:
    time_handler()
    {
        handle_open = boost::bind(                          //处理打开
            &time_handler::time_handle_open, this, _1);
        handle_write = boost::bind(                         //处理写
            &time_handler::time_handle_write,this, _1, _2);
    }
private:
```

```
    void time_handle_open(tcp_session_type session)
    {
        LOG_TRACE("time start from:");
        LOG_TRACE(session->socket().remote_endpoint().address());

        int32_t t = boost::asio::detail::socket_ops::  //获得秒数
            host_to_network_long(static_cast<int>(time(0)));

        session->write(&t, sizeof(t));
    }
    void time_handle_write(tcp_session_type session, std::size_t n)
    {
        LOG_TRACE("write complete:" << n);
        session->close();
    }
};                                              // time _handler 定义结束
```

chargen 协议

chargen 协议不停地向客户端发送数据，而不关心客户端发送的数据，所以我们需要在 handle_open 时就发送数据，并在数据发送完毕后（handle_write）持续不断地发送：

```
#include <boost/circular_buffer.hpp>
#include <boost/next_prior.hpp>
#include <boost/iterator/counting_iterator.hpp>

class chargen_handler : public tcp_handler       //继承 tcp_handler
{
private:
    boost::circular_buffer<char> m_msg;           //循环缓冲容器
    void init()                                   //向 m_msg 添加字符
    {
        std::copy(                                //使用 copy 算法，搭配计数迭代器
            boost::make_counting_iterator(0x20)),  //从空格开始
            boost::make_counting_iterator(0x7f),   //直至~结束
            std::back_inserter(m_msg)  );          //添加到循环缓冲区
    }
public:
    chargen_handler(): m_msg(0x7f - 0x20))        //存储从' '到'~'的所有字符
    {
        init();                                   //向 m_msg 添加字符
```

```
        handle_open = boost::bind(                    //处理打开
            &chargen_handler::chargen_handle_open, this, _1);
        handle_write = boost::bind(                   //处理写
            &chargen_handler::chargen_handle_write,this, _1, _2);
    }
private:
    void chargen_handle_open(tcp_session_type session)
    {
        LOG_TRACE("chargen start from:");
        LOG_TRACE(session->socket().remote_endpoint().address());

        write(session);                               //发送数据
    }
    void chargen_handle_write(tcp_session_type session, std::size_t n)
    {
        LOG_TRACE("write complete:" << n);
        write(session);                               //继续发送数据
    }

    void write(tcp_session_type session)              //发送数据
    {
        //从循环缓冲区头拷贝 72 个字符
        std::vector<char> tmp(m_msg.begin(), m_msg.begin() + 72);

        //把字符加入 TCP 发送缓冲，末尾附加回车符
        session->write_buf().append(&tmp[0], tmp.size());
        session->write_buf().append("\n", 1);
        session->write();                             //发送

        m_msg.rotate(boost::next(m_msg.begin()));    //循环缓冲旋转
    }
};      // chargen_handler 定义结束
```

chargen_handler 中使用了 Boost 库的新式容器 circular_buffer，它是一个环形可复用的容器，我们使用 3.6.5 小节的计数迭代器非常容易地生成了所有的 ASCII 码可打印字符来初始化它。

在发送数据时我们从 circular_buffer 的头部取出 72 个字节，然后调用它的 rotate()方法让环形容器旋转，这样就得到了内容不断变化的新消息。注意这里不能使用 circular_buffer 的 linearize()方法，它只是简单地把容器内部元素线性化，不具有

环绕的能力[①]。

服务器

实现了这些回调函数类后，现在我们使用模板类形式的 tcp_server 来启动这些服务，并使用一个外部的 io_service_pool，这样在一个 main() 中就可以启动多个服务：

```
int main()
{
  log_init();                                           //初始化日志库
  cout << "server start" << endl;

  io_service_pool ios;                                  //并发线程池

  tcp_server<echo_handler> echo_svr(ios, 7);            //echo 协议
  tcp_server<discard_handler> discard_svr(ios, 9);      //discard 协议
  tcp_server<daytime_handler> daytime_svr(ios, 13);     //daytime 协议
  tcp_server<chargen_handler> chargen_svr(ios, 19);     //chargen 协议
  tcp_server<time_handler> time_svr(ios, 37);           //time 协议

  ios.run();                                            //启动并发服务
};
```

这些协议对应的客户端代码都比较简单，这里就不提供了，读者可自行验证。

12.2.7　HTTP 协议的实现

使用回调函数我们也可以轻松实现 HTTP 协议，开发出一个轻量级的 HTTP 服务器（当然不能和 apache、nginx 这些顶级的服务器相比）。由于 HTTP 协议是基于纯文本的，所以可以使用 boost.xpressive 库提供的正则表达式功能来解析 HTTP 请求，访问磁盘上的文件则可以使用 boost.filesystem。

本小节不讨论如何完整地实现 HTTP 协议，只给出一个最简单的 HTTP 处理类，它不解析请求，只返回一个固定长度的字符串用于测试：

```
class http_handler : public tcp_handler  //http 协议简单实现
{
public:
```

① 这里也可以使用标准容器搭配算法 rotate，读者可自行尝试。

```
    http_handler()                              //构造函数，只处理读写回调
    {
        handle_read = boost::bind(
            &http_handler::http_handle_read,this, _1, _2);
        handle_write = boost::bind(
            &http_handler::http_handle_write,this, _1, _2);
    }
private:
    void http_handle_read(tcp_session_type session, std::size_t n)
    {
        string content(1024, 'a');              //返回 1024 个字符
        string str =                            //组装 http 响应头
            "HTTP/1.0 200 OK\r\n"               //状态行
            "Content-Length: 1024\r\n"          //响应报头
            "Content-Type: text/plain\r\n"
            "\r\n"  ;
        str += content;                         //响应正文

        session->write(str.c_str(), str.size());  //发送给客户端
    }
    void http_handle_write(tcp_session_type session, std::size_t n)
    {
        session->close();                       //发送完毕后关闭连接
    }
};                                              // http_handler 定义结束
```

这样我们就成功地实现了一个最简单的 HTTP 服务器，主程序如下：

```
int main()
{
    cout << "http server start" << endl;        //调试信息

    tcp_server<http_handler> svr(6677,2);       //使用 6677 端口提供 http 服务
    svr.run();      //使用内部的 io_service_pool 启动服务器
}
```

启动这个 HTTP 服务器后可以使用命令行工具 curl（Mac、Linux 共通）或者 wget（Linux）[1]对它进行验证。

① 这两个都是功能强大的命令行下载工具，相对来说 curl 功能更强、支持的协议更多一些。

```
curl -o a.txt http://127.0.0.1:6677
```

会把服务器传回的字符串保存为 a.txt 文件，cat 命令查看文件会看到一堆“a”字符。

在作者的 Mac（intel core2 duo 2.4GB）上使用 ab（ApacheBench）对这个 HTTP 服务器执行了简单的性能测试，结果如下：

```
$ ab -c 100 -n 10000 'http://127.0.0.1:6677/'    #测试并发一万个请求

Concurrency Level:    100
Time taken for tests: 1.026 seconds
Complete requests:    10000
Failed requests:      0
Write errors:         0
Total transferred:    11004917 bytes
HTML transferred:     10329088 bytes
Requests per second:  9746.17 [#/sec] (mean)
Time per request:     10.260 [ms] (mean)
Time per request:     0.103 [ms] (mean, across all concurrent requests)
Transfer rate:        10474.20 [Kbytes/sec] received

Connection Times (ms)
             min  mean[+/-sd] median   max
Connect:     0    2      1.3   2        11
Processing:  0    8      3.6   7        41
Waiting:     0    6      3.0   6        26
Total:       0    10     3.3   9        41

Percentage of the requests served within a certain time (ms)
  50%      9
  66%     10
  75%     11
  80%     12
  90%     14
  95%     15
  98%     19
  99%     24
 100%     41 (longest request)
```

当然这个性能测试结果并不能说明什么问题，因为我们的 HTTP 服务器没有实现任何功能逻辑，有兴趣的读者可以在此基础上进一步完善 HTTP 功能。

12.3 多线程工具

现在计算机硬件早已经进入了多核心的时代，而对于服务器网络编程来说，多线程更是开发高性能程序的必需手段，多线程编程对于广大程序员来说已经是一个无法回避的问题。不同的操作系统都提供各自的多线程 API，也存在很多封装这些原始 API 的多线程库，但 boost.thread 无疑是其中最容易使用的一个。

boost.thread 库里包含多线程编程所需的几乎所有概念，如线程/线程池、互斥量、读写锁、条件变量、barrier（护栏、倒计数锁）、线程局部存储等等，基于它们我们可以任意编写所需的多线程构件（有关它的详细论述可见推荐书目[1]）。

本小节中我们来实现五个较为常用的多线程工具类：

- job_queue ：一个可用于多线程环境下的阻塞队列，可以用于生产者-消费者模式；
- worker ：使用 job_queue 的一个具体的消费者，内部可以有多个线程；
- scheduler ：使用睡眠功能实现的简单线程调度器；
- safe_map ：一个线程安全的映射容器，使用了散列容器 boost::unorederd_map 作为内部实现，因而更加高效；
- safe_singleton：一个线程安全的单件。

12.3.1 job_queue

job_queue 是一个很简单但却很有用的工具，它使用条件变量作为管理并发的手段，可以同时被多个线程使用。为了使它能够容纳任意的数据，job_queue 应该被实现为一个模板类。

包含头文件

我们决定使用标准容器 std::deque 作为 job_queue 的内部实现，需要包含头文件如下：

```
// job_queue.hpp [2011 Aug chrono]
#include <deque>                        //标准容器

#include <boost/noncopyable.hpp>        //boost 基本工具
```

```
#include <boost/utility/value_init.hpp>
#include <boost/assert.hpp>
#include <boost/concept_check.hpp>

#define BOOST_ALL_NO_LIB
#include <boost/thread/mutex.hpp>                               //多线程工具
#include <boost/thread/condition_variable.hpp>
```

代码实现

job_queue 使用 boost::noncopyable 来禁止对它的拷贝，使用概念检查（10.3 小节）保证容器容纳元素的正确性：

```
template <typename Job>                                         //模板参数
class job_queue : boost::noncopyable                            //禁止拷贝
{
public:
  typedef Job job_type;                                         //元素类型定义
  typedef std::deque<job_type> queue_type;                      //容器类型定义

  typedef boost::mutex mutex_type;                              //互斥量类型定义
  typedef typename mutex_type::scoped_lock lock_type;           //锁定义
  typedef boost::condition_variable_any condition_type;         //条件变量定义

  BOOST_CONCEPT_ASSERT((boost::SGIAssignable<job_type>));//概念检查
  BOOST_CONCEPT_ASSERT((boost::DefaultConstructible<job_type>));

private:
  queue_type m_queue;                                           //内部容器

  mutex_type m_mutex;                                           //互斥量定义
  condition_type m_hasJob;                                      //条件变量定义

  bool m_stop_flag;                                             //一个用于停止工作的标志
```

在 job_queue 的内部我们同样没有直接使用模板参数 Job、deque 和 boost::mutex 等类型，而是使用 typedef 重新定义了若干个内部类型。

job_queue 的接口很简单，就是任务的入队和出队，以及在某些必要的情况下停止工作：

```
public:
  job_queue():m_stop_flag(false)                                //构造函数
```

```
    {  }

    void push(const job_type& x)                  //任务入队
    {
        lock_type lock(m_mutex);                  //锁定互斥量
        m_queue.push_back(x);                     //入队
        m_hasJob.notify_one();                    //只通知一个等待的线程
    }

    job_type pop()                                //任务出队
    {
        lock_type lock(m_mutex);                  //锁定互斥量
        while(m_queue.empty() && !m_stop_flag)    //当队列空且未要求停止时
        {   m_hasJob.wait(m_mutex);}              //条件变量等待

        if (m_stop_flag)                          // 强制停止队列
        {
            return boost::initialized_value;      //相当于 return job_type()
        }

        BOOST_ASSERT(!m_queue.empty());           //断言队列中有元素

        job_type tmp = m_queue.front();           //取队列前端元素
        m_queue.pop_front();

        return tmp;                               //返回取出的元素
    }

    void stop()                                   //停止队列的工作
    {
        m_stop_flag = true;                       //设置停止标志
        m_hasJob.notify_all();                    //通知所有等待的线程
    }
};//job_queue 定义结束
```

job_queue 的实现代码比较简单，只有 pop()函数略微复杂些，它需要检测停止标志变量 m_stop_flag，如果外界要求停止队列的工作，那么它就退出等待。因为之前我们已经使用概念检查保证了 job_type 是可缺省构造且可赋值的，所以我们可以直接使用 value_initialized 库（参见 2.4 小节）的 initialized_value 对象返回一个空的 job_type 对象，相当于调用 job_type()。

单元测试

job_queue 的简单单元测试代码如下：

```
void test_queue()
{
  job_queue<int> q;               //使用 int 作为队列元素

  q.push(10);                     //入队
  q.push(20);                     //入队

  int tmp = q.pop();              //出队
  BOOST_TEST_EQ(tmp, 10);
  BOOST_TEST_EQ(q.pop(), 20);     //出队

  q.push(30);                     //入队
  q.stop();                       //停止队列
  tmp = q.pop();                  //出队
  BOOST_TEST_EQ(tmp, 0);          //元素应该是 0
}
```

12.3.2 worker

worker 是一个工作在 job_queue 上的消费者，它可以使用一个或多个线程获取 job_queue 中的数据然后处理。同样的，我们把它也实现为模板类。

包含头文件

worker 包含的头文件如下：

```
// worker.hpp [2011 Aug chrono]
#include <boost/assert.hpp>
#include <boost/bind.hpp>         //boost.bind
#include <boost/function.hpp>     //boost.function

#define BOOST_ALL_NO_LIB
#include <boost/thread.hpp>       //线程库

#include "job_queue.hpp"          //任务队列
```

代码实现

worker 不使用模板方法模式（即虚函数）来定义具体的处理工作，而是使用 boost.function 存储处理函数(即回调)，这样用起来更加灵活，执行效率更高，多线程方面则还是使用 boost::thread_group 集中管理：

```
template<typename Queue>
class worker
{
public:
    typedef Queue queue_type;                       //内部类型定义
    typedef typename Queue::job_type job_type;      //使用 job_queue 的 traits
    typedef boost::function<bool(job_type&)> func_type;//处理函数类型

private:
    queue_type&         m_queue;                    //关联的任务队列
    func_type           m_func;                     //处理函数，处理任务

    int                 m_thread_num;               //线程数量
    boost::thread_group m_threads;                  //线程池
```

worker 有两个构造函数，第一种形式的构造函数是模板函数，可以接受函数指针或函数对象作为回调处理，第二种形式的构造函数不传递回调函数，回调函数由之后的 start() 传入：

```
public:
    template<typename Func>                         //模板参数是一个可调用物
    worker(queue_type& q, Func func, int n = 1):
        m_queue(q), m_func(func),
        m_thread_num(n)
    { BOOST_ASSERT(n > 0);}                         //断言至少有一个线程

    worker(queue_type& q,int n = 1):                //暂不传递回调函数
        m_queue(q), m_thread_num(n)
    { BOOST_ASSERT(n > 0);}                         //断言至少有一个线程
```

worker 的主要功能函数是 start() 和 run()，它们分别启动多个工作线程处理 job_queue 里的数据，两者的区别在于 start() 是不阻塞的而 run() 是阻塞的，与 io_service_pool 一样：

```
public:
  void start()                         //启动多个工作线程，非阻塞，不传递回调函数
  {
      BOOST_ASSERT(m_func);            //断言已经设置了回调函数
      if (m_threads.size() > 0)        //已经启动了线程
      {   return; }

      for (int i = 0;i < m_thread_num; ++i)//启动n个线程
      {
          m_threads.create_thread(     //线程池创建多个线程
              boost::bind(&worker::do_work, this));//使用bind
      }
  }

  template<typename Func>              //模板参数是一个可调用物
  void start(Func func)                //设置回调函数并启动线程
  {
      m_func = func;                   //设置回调函数
      start();                         //启动线程
  }

  void run()                           //启动多个工作线程，阻塞
  {
      start();                         //启动线程
      m_threads.join_all();            //阻塞等待
  }
```

读者需要注意 start()函数中的创建线程代码，我们使用了 boost.bind，绑定了工作函数 do_work()，它执行真正的处理数据工作：

```
private:
  void do_work()                       //循环处理工作队列
  {
      for(;;)                          //无限循环
      {
          job_type job = m_queue.pop(); //从队列取出一个任务

          if (!m_func || !m_func(job))  //使用处理函数处理任务
          {   break;}                   //当m_func为空或处理出错时停止循环
      }   //end for
  }
```

最后是一个停止线程工作的函数 stop()，它只是简单地把处理函数置空：

```
public:
  void stop()
  {
      m_func = 0;                   //置空处理函数，相当于调用 clear()
      m_queue.stop();               //停止队列
  }
};                                  //worker 定义结束
```

理论上来说使用 boost::thread_group 的成员函数 interrupt_all() 强制终止线程也是可行的，但这样的做法太过“粗暴”，我们最好是让线程自己安全地停止[①]。

单元测试

worker 的简单测试代码如下：

```
bool func_int(int& x)               //处理函数
{
  cout << "work on : " << x << endl; //简单打印整数的值
  return true;
}

void test_worker()
{
  typedef job_queue<int> queue_type; //任务队列定义
  queue_type q;

  q.push(10);                       //入队
  q.push(20);

  worker<queue_type> w(q, func_int); //worker 对象
  w.start();                        //启动工作线程
  this_thread::sleep(posix_time::milliseconds(100)); //等待处理完成

  q.push(30);                       //入队
  this_thread::sleep(posix_time::milliseconds(100)); //等待处理完成
```

① 这里有个小问题，如果有多个 worker 均在一个 job_queue 上工作，这时调用 stop() 会使此 job_queue 上的所有 worker 均停止工作。但如果不停止 job_queue 那么 worker 可能会永远阻塞在 m_queue.pop() 处无法结束。

```
    w.stop();停止工作线程
    this_thread::sleep(posix_time::milliseconds(100)); //等待处理完成
}
```

12.3.3 scheduler

scheduler 类似 asio 的定时器 deadline_timer，可以定时执行特定的功能，但它不需要使用 asio 的异步机制，而是使用了 thread 库的睡眠功能，用起来要方便一些。

包含头文件

scheduler 包含的头文件如下：

```
#include <vector>

#include <boost/bind.hpp>
#include <boost/function.hpp>
#include <boost/foreach.hpp>

#define BOOST_ALL_NO_LIB
#include <boost/date_time/posix_time/posix_time.hpp> //日期时间库
#include <boost/thread.hpp>                          //线程库
```

代码实现

scheduler 同样使用了 boost::function 来存储被调用的函数，用 boost::thread_group 管理多个线程：

```
class scheduler                                         //多线程实现的定时器
{
public:
    typedef boost::function<void()> func_type;          //函数类型定义
    typedef std::pair<func_type, int> schedule_type;    //存储调用间隔时间
    typedef std::vector<schedule_type> schedule_funcs; //容器类型定义
private:
    schedule_funcs m_funcs;                             //用 vector 存储多个回调函数
    boost::thread_group m_threads;                      //线程组
```

scheduler 的接口同样很小，用一个 add() 函数添加要执行的回调函数，start() 和 run() 启动线程：

```
public:
  void add(func_type func, int seconds)          //传递回调函数和间隔时间
  {
      m_funcs.push_back(std::make_pair(func, seconds));     //添加到容器
  }

  void start()                                          //启动线程，非阻塞
  {
      BOOST_FOREACH(schedule_funcs::reference x, m_funcs)    //遍历容器
      {
          m_threads.create_thread(                           //启动线程
              boost::bind(&scheduler::schedule, this,
                  x.first, x.second));
      }
  }

  void run()                                       //启动线程并阻塞等待
  {
      start();
      m_threads.join_all();
  }
```

start()函数中调用了私有成员函数 schedule()，它使用一个循环执行回调函数，并在执行后睡眠若干秒：

```
private:
  void schedule(func_type func, int sec)       //睡眠调度执行
  {
      for (;;)
      {
          func();                              //执行函数
          boost::this_thread::sleep(            //睡眠若干秒
                      boost::posix_time::seconds(sec));
      }
  }
};                                             //scheduler 定义结束
```

可以使用一个很小的测试函数来验证 scheduler 的功能：

```
void test_scheduler()     //验证 scheduler 的功能
{
```

```
    scheduler s;                                //scheduler 对象

    int x = 10;
    s.add(boost::bind(func_int, x), 3);         //使用之前的 func_int 函数
    s.run();                                    //启动定时器，每 3 秒执行一次
}
```

12.3.4 safe_map

STL 容器和 Boost 容器大多不提供完全的线程安全，在多线程环境下使用可能会发生意想不到的错误，因此有必要对它们加以封装，增加线程安全，这需要使用 thread 库的共享互斥量 shared_mutex（又称读写锁）。

包含头文件

safe_map 封装了 boost::unordered_map，因而可以高效地检索数据，需包含的头文件如下：

```
// safe_map.hpp [2011 Aug chrono]
#include <algorithm>                            //标准算法库

#include <boost/noncopyable.hpp>                //boost 基本工具
#include <boost/assert.hpp>
#include <boost/unordered_map.hpp>

#define BOOST_ALL_NO_LIB
#include <boost/thread/shared_mutex.hpp>        //多线程共享互斥量
#include <boost/thread/locks.hpp>
```

代码实现

为了方便使用，safe_map 的模板参数做了适当的简化，只有 Key 和 Value 两个参数：

```
template <typename Key, typename Value>
class safe_map : boost::noncopyable                     //线程安全读写的 hash_map
{
  typedef boost::unordered_map<Key, Value> map_type;   //使用散列表

  typedef boost::shared_mutex            rw_mutex;     //共享互斥量定义
  typedef boost::shared_lock<rw_mutex> read_lock;      //读锁定
  typedef boost::unique_lock<rw_mutex> write_lock;     //写锁定
```

```
public:
  //容器必须的 typedef
  typedef typename map_type::key_type key_type;
  typedef typename map_type::mapped_type  mapped_type;
  typedef typename map_type::value_type value_type;

  typedef typename map_type::pointer pointer;
  typedef typename map_type::const_pointer const_pointer;

  typedef typename map_type::size_type size_type;
  typedef typename map_type::difference_type difference_type;
  typedef typename map_type::reference reference;
  typedef typename map_type::const_reference const_reference;
  typedef typename map_type::iterator    iterator;
  typedef typename map_type::const_iterator  const_iterator;

private:
  map_type         m_map;                //内部容器
  mutable          rw_mutex m_mutex;     //共享互斥量，应该是 mutable 的
```

注意，safe_map 的代码中有大量的 typedef，它们是为了满足容器的概念检查而提供的，可以让这个容器封装类像真正的容器一样使用—比如用于 BOOST_FOREACH。

safe_map 提供了若干与标准容器一致的操作接口，但没有提供所有接口（可以根据需要后续添加）。每个操作首先视情况调用读锁或写锁，然后把请求转发给 boost::unordered_map：

```
public:
  bool empty() const                     //判断容器是否为空
  {
      read_lock lock(m_mutex);
      return m_map.empty();
  }
  size_type size() const                 //获取容器的大小
  {
      read_lock lock(m_mutex);
      return m_map.size();
  }
  size_type max_size() const             //获取容器的最大可能大小
```

```
    {
        read_lock lock(m_mutex);
        return m_map.max_size();
    }

    iterator begin()                                    //迭代器接口
    {
        read_lock lock(m_mutex);
        return m_map.begin();
    }
    const_iterator begin() const
    {
        read_lock lock(m_mutex);
        return m_map.begin();
    }
    const_iterator end() const
    {
        read_lock lock(m_mutex);
        return m_map.end();
    }
    iterator end()
    {
        read_lock lock(m_mutex);
        return m_map.end();
    }

    bool insert(const key_type& k, const mapped_type& v) //添加元素
    {
        write_lock lock(m_mutex);
        return m_map.insert(value_type(k, v)).second;  //返回 bool 表示是否成功
    }

    bool find(const key_type& k)                        //查找元素
    {
        read_lock lock(m_mutex);
        return m_map.find(k) != m_map.end();
    }

  size_type erase(const key_type& k)                    //删除元素
    {
```

```
        write_lock lock(m_mutex);
        return m_map.erase(k);
    }

    void clear()                                        //清空容器
    {
        write_lock lock(m_mutex);
        m_map.clear();
    }
```

safe_map 的 insert() 和 find() 函数较标准接口有所变动，主要是出于方便使用的目的，特别是 find()，它必须在一个锁定之内完成与逾尾迭代器的比较操作，否则在多线程环境下的迭代器比较结果可能会不一致。

但在实际使用时我们不能依赖 find() 的结果，因为在多线程的环境下即使本次操作找到了元素，下一次操作元素也有可能被其他的线程修改或删除，所以访问元素操作我们改用一个返回 optional 对象的特殊 at() 函数，在一个读锁之内完成查找和取值的操作，根据 optional 对象是否有效来判断值是否存在再使用：

```
typedef boost::optional<mapped_type> optional_mapped_type;
optional_mapped_type at(const key_type& k)           //访问元素
{
    read_lock lock(m_mutex);
    if (m_map.find(k) != m_map.end())                 //元素存在
    { return optional_mapped_type(m_map[k]);}

    return optional_mapped_type();
}
```

safe_map 仍然有 operator[]，但它必须是只读的，修改值要用一个新的 set() 函数：

```
const mapped_type& operator[](const key_type& k)     //访问元素
{
    read_lock lock(m_mutex);
    BOOST_ASSERT(m_map.find(k) != m_map.end());       //元素必须已经存在
    return m_map[k];
}
void set(const key_type& k, const mapped_type& v)    //修改元素
{
    write_lock lock(m_mutex);                         //写锁定
```

```
        m_map[k] = v;                              //新增或修改元素
    }
```

为了能够线程安全地操作容器内的所有元素，safe_map 还提供了一个特别的 for_each 操作，它使用 read_lock 锁住容器，然后用标准算法 for_each 遍历操作容器[1]。

```
    template<typename Func>
    void for_each(Func func)        //传递一个操作容器内元素的函数或函数对象
    {
        read_lock lock(m_mutex);
        std::for_each(              //标准 for_each 算法
            m_map.begin(), m_map.end(), func);
    }
};//safe_map 定义结束
```

单元测试

safe_map 的简单测试代码如下：

```
void test_safemap()
{
    typedef safe_map<int,string> map_type;          //类型定义
    BOOST_CONCEPT_ASSERT((Container<map_type>));    //概念检查
    BOOST_CONCEPT_ASSERT((ForwardContainer<map_type>));

    map_type m;
    BOOST_TEST(m.empty());

    m.insert(10, "ten");                            //添加元素
    m.set(20, "twenty");

    BOOST_TEST(!m.empty());
    BOOST_TEST_EQ(m.size(), 2);

    BOOST_TEST(m.find(10) && m.find(20));           //查找元素
    BOOST_TEST(!m.find(50));
    BOOST_TEST(!m.insert(10, "one"));

    typedef map_type::value_type v_t;
```

① 这里我们也可以使用 Boost 库的 foreach 算法。

```
    BOOST_FOREACH(v_t& v, m)                    //使用 boost.foreach 算法
    {
        cout << v.first << "-"
            << v.second << endl;
    }

    BOOST_TEST_EQ(m.erase(10), 1);              //删除元素
    BOOST_TEST(!m.find(10));

    m.clear();                                  //清空容器
    BOOST_TEST(m.empty());
}
```

读者可以仿照 safe_map 的实现手法自行编写其他 STL 容器或 Boost 容器的多线程封装。

12.3.5 safe_singleton

Boost 程序库目前没有专门的单件库,虽然在 pool 库和 serialization 库里有两个实现，但用起来总是不太方便，因此我们有必要自行实现一个线程安全的单件类。

safe_singleton 的实现原理很简单，它的基本结构与标准单件差不多，只是使用了 thread 库的 call_once()函数来保证单件只调用 init()函数初始化一次：

```
// safe_singleton.hpp [2011 Aug chrono]
#include <boost/noncopyable.hpp>
#define BOOST_THREAD_NO_LIB
#include <boost/thread/once.hpp>                //  call_once()

template<typename T>
class safe_singleton: boost::noncopyable //线程安全的单件类
{
public:
    typedef T real_type;                        //单件的 traits 定义
    static real_type& instance()                //访问单件的实例
    {
        boost::call_once(m_flag, init);         //只调用 init()函数一次
        return init();                          //返回单件
    }
```

```
private:
  static real_type& init()          //初始化函数
  {
      static real_type obj;         //静态局部变量
      return obj;                   //返回单件
  }

  static boost::once_flag m_flag;   //初始化标志，静态成员变量
};                                  //safe_singleton 定义结束

//初始化静态成员变量
template<typename T>
boost::once_flag safe_singleton<T>::m_flag = BOOST_ONCE_INIT;
```

在 safe_singleton 的实现里我们同样使用了 traits 的手法，不是直接使用模板参数 T，而是定义了内部类型 real_type，这样无论何时我们总能够通过它来获得单件的真正类型信息。

12.4 第二个 TCP 服务器

12.2 小节我们实现了第一个 TCP 服务器，但它只用了很少的多线程功能，仅仅是把 io_service 运行在不同的线程中，实质上还是单线程的并发处理。本小节我们实现另外一个 TCP 服务器，它使用了 12.3 小节定义的多线程工具，处理常用的“消息头+消息体”格式的数据。

这个 TCP 服务器的设计结合了事件驱动模型和 SEDA 模型，使用工作队列交换、处理消息，分离了各个功能模块，好处是逻辑清晰，结构灵活，能够充分发挥多线程的优势。

同之前一样，先简单介绍一下服务器程序中的各个类：

- tcp_message：对消息格式的封装，固定大小；
- tcp_session：读写 TCP 数据流，把收到的消息存入队列；
- tcp_server ：侦听端口创建 TCP 连接。

这三个类及关联类的 UML 描述如图 12-2 所示：

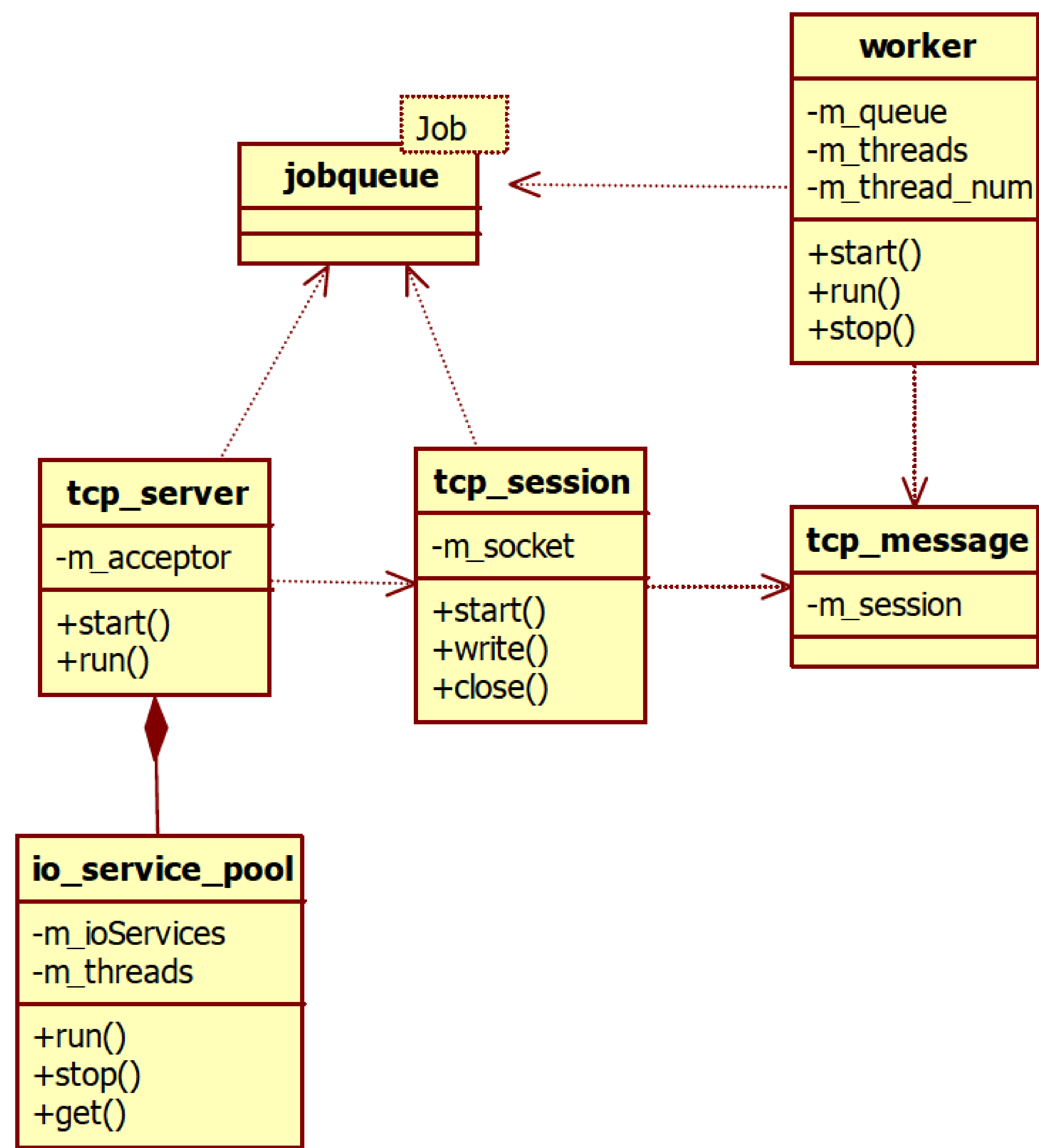


图 12-2 第二个 TCP 服务器的类图

12.4.1 消息结构定义

本节中我们处理的是特定结构的消息，一条完整的消息包括消息头和消息体两部分，结构如下所示：

消息头（12 字节）	消息体（最大长度 1024 字节，可自定义）

其中的消息头又可以再细化，由三个 32 位的整数构成：

消息类型（整数）	消息体大小（整数）	CRC 校验（整数）

把以上定义用 C++语言表述成源代码的形式就是：

```
// tcp_msg_def.h [2011 Aug chrono]
#include "intdef.h"                //标准整数定义

struct msg_head                    //消息头结构定义
```

```
{
  uint32_t type;                          //消息类型
  uint32_t size;                          //消息体大小
  uint32_t chksum;                        //CRC 校验
};
#define MAX_BODY_SIZE 1024                //消息体最大长度，可根据需要变动
```

对于长度超过 MAX_BODY_SIZE 的消息我们可以把它分割成若干条较短的消息，再使用这个消息结构发送。

12.4.2　tcp_message

tcp_message 封装了消息结构的处理，需要包含的头文件如下：

```
// tcp_message.hpp [2011 Aug chrono]
#include "tcp_msg_def.h "                 //消息结构定义

#include <boost/checked_delete.hpp>       //带检查的指针删除
#include <boost/smart_ptr.hpp>            //智能指针
#include <boost/array.hpp>                //数组的 STL 容器风格包装
#include <boost/function.hpp>             //boost::function
#include <boost/noncopyable.hpp>          //禁止拷贝
#include <boost/crc.hpp>                  //crc 校验
```

因为消息的最大长度是固定的，为了避免消息在传递过程中反复拷贝的代价，我们完全可以把 tcp_message 设计为 noncopyable 的，强制它只能以指针的方式使用，而它的创建方式则可以灵活处理—既可以直接用 new 在堆上创建，也可以使用内存池。为了能够自如地释放，tcp_message 模仿了 shared_ptr，要求在构造函数里提供一个销毁器，可以自主销毁[①]。

tcp_message 中除了保存基本的消息头和消息体数据外，还使用了一个 shared_ptr 来保管 tcp_session，这样就把纯数据的消息与具有收发功能的 TCP 连接联系了起来（类似 Command 模式）。

tcp_message 的成员变量定义如下：

```
class tcp_message : boost::noncopyable    //不允许拷贝
{
```

① 我们也可以把 tcp_message 的析构函数设置为私有的，强制它只能在堆上创建并使用成员函数销毁，但这样就不能够使用内存池了。

```
public:
  typedef boost::function<void(tcp_message*)> destory_type;//销毁器
  typedef char char_type;               //消息的字符表示类型

private:
  tcp_session_ptr m_session;            //tcp 连接的 shared_ptr
  destory_type    m_destory;            //用于自行销毁
```

对应这两个成员变量的构造函数和访问函数代码如下，其中构造函数有两个重载形式，如果不传递销毁器，那么 tcp_message 将默认使用 checked_delete 删除 this 指针：

```
public:
  template<typename Func>               //模板参数是销毁器
  tcp_message(const tcp_session_ptr& s, Func func)://构造函数
      m_session(s), m_destory(func)
  {}

  tcp_message(const tcp_session_ptr& s): //另一个构造函数
      m_session(s)                      //不传递销毁器
  {}

  tcp_session_ptr get_session()         //获得 tcp_session 的智能指针
  {   return m_session;}

  void destory()                        //销毁消息自身
  {
      if (m_destory)
      {   m_destory(this);    }         //有销毁器则调用销毁器
      else
      {   boost::checked_delete(this);} //使用 checked_delete 销毁
  }
```

消息头和消息体是两块（也可以合并成一块）固定大小的内存，为了方便起见我们用 boost::array 来实现，并提供两个可直接访问的成员函数：

```
public:
  typedef boost::array<char_type, sizeof(msg_head)> head_data_type;
  typedef boost::array<char_type, MAX_BODY_SIZE> body_data_type;
private:
  head_data_type m_head;                //消息头
  body_data_type m_msg;                 //消息体
```

```
public:
  head_data_type& head_data()              //获得消息头数据
  {   return m_head;  }

  body_data_type& msg_data()               //获得消息体数据
  {   return m_msg;}
```

目前的消息结构很简单，所以对它们的操作也不多，而且都很简单：

```
public:
  msg_head* get_head()                     //转换为消息头结构操作
  {   return reinterpret_cast<msg_head*>(m_head.data());}

  bool check_head()                        //简单地检查消息头是否正确
  {   return (get_head()->size < MAX_BODY_SIZE);}

  bool check_msg_crc()                     //检查消息体的 crc 校验
  {
      boost::crc_32_type crc32;            //使用 boost 的 crc 库
      crc32.process_bytes(&m_msg[0], get_head()->size);

      return get_head()->chksum == crc32.checksum(); //比较 crc 值
  }

  void set_msg_crc()                       //在消息头里设置消息体的 crc 校验
  {
      boost::crc_32_type crc32;            //使用 boost 的 crc 库
      crc32.process_bytes(&m_msg[0], get_head()->size);

      get_head()->chksum = crc32.checksum();         //设置 crc 值
  }
};                                         //tcp_message 定义结束
```

上面的代码中 check_msg_crc()和 set_msg_crc()的代码很相似，仅有微小的不同，请读者注意。

最后，我们为 tcp_message 定义一些意义更明确、更便于使用的 typedef：

```
typedef tcp_message   tcp_request;         //tcp 请求消息
typedef tcp_message   tcp_response;        //tcp 响应消息
typedef tcp_request*  tcp_request_ptr;     //tcp 请求消息指针
typedef tcp_response* tcp_response_ptr;    //tcp 响应消息指针
```

12.4.3 tcp_session

本小节中的 tcp_session 与 12.2.3 小节的 tcp_session 实现原理基本相同，但因为要适应多线程环境和处理消息，所以也有一些不同。

头文件

tcp_session.h 需包含如下的头文件：

```
// tcp_session.h [2011 Aug chrono]
#include "tcp_message.hpp"                        //消息定义
#include "job_queue.hpp"                          //工作队列

#include <boost/smart_ptr.hpp>                    //智能指针
#include <boost/enable_shared_from_this.hpp> //从 this 创建 shared_ptr
#include <boost/pool/object_pool.hpp>             //对象池

#define BOOST_ALL_NO_LIB
#include <boost/asio.hpp>                         //boost.asio 库
```

tcp_session 的内部类型定义和成员变量如下：

```
class tcp_session :
  public boost::enable_shared_from_this<tcp_session>
{
public:
  typedef boost::asio::ip::tcp::socket socket_type;  //内部类型定义
  typedef boost::asio::io_service ios_type;

  typedef ios_type::strand strand_type;              //asio 的 strand 概念
  typedef job_queue<tcp_request_ptr> queue_type;     //消息队列
  typedef boost::object_pool<tcp_message> object_pool_type;//内存池
private:
  socket_type                m_socket;          //asio 的 socket 封装

  strand_type                m_strand;          //asio 的 strand 概念
  queue_type&                m_queue;           //消息队列
  static object_pool_type m_msg_pool;        //内存池
```

与 12.2.3 小节相比，tcp_session 多了下面三个新的类型和成员变量：

- strand_type ：asio 库提供的线程概念，可以包装异步操作在多线程环境里安全地执行；
- queue_type ：接受 tcp_message 的消息队列，队列里保存消息的指针而不是拷贝；
- object_pool_type：使用 pool 库的一个内存池，用来分配 tcp_message。

tcp_session 的成员函数如下：

```
public:
  tcp_session(ios_type& ios, queue_type& q); //构造函数

  socket_type& socket();                     //获得 socket
  ios_type& io_service();                    //获得 io_service

  void start();                              //启动 TCP 连接，开始读数据
  void close();                              //关闭 TCP 连接
  void write(tcp_response_ptr resp);         //发送消息

private:
  tcp_request_ptr create_request();          //创建消息对象
  void read(tcp_request_ptr req);            //接收消息

  void handle_read_head(const boost::system::error_code& error,
      size_t bytes_transferred,
      tcp_request_ptr req);                  //消息头读处理函数
  void handle_read_msg(const boost::system::error_code& error,
      size_t bytes_transferred,
      tcp_request_ptr req);                  //消息体读处理函数

  void handle_write_head(const boost::system::error_code& error,
      size_t bytes_transferred,
      tcp_response_ptr resp);                //消息头写处理函数
  void handle_write_msg(const boost::system::error_code& error,
      size_t bytes_transferred,
      tcp_response_ptr resp);                //消息体写处理函数
};     //tcp_session 定义结束
```

构造函数和成员访问函数

在 tcp_session 的实现文件中我们首先要定义静态成员变量 m_msg_pool：

```
#include "tcp_session.h"
#include <boost/bind.hpp>
using namespace boost;
using namespace boost::asio;

tcp_session::object_pool_type tcp_session::m_msg_pool;      //内存池
```

tcp_session 的构造函数和成员访问函数很简单，注意在构造函数中我们是如何初始化 m_strand 和 m_queue 的：

```
tcp_session::tcp_session( ios_type& ios, queue_type& q ):   //构造函数
  m_socket(ios), m_strand(ios), m_queue(q)                  //初始化成员变量
{}

tcp_session::socket_type& tcp_session::socket()             //获得 socket
{ return m_socket;}

tcp_session::ios_type& tcp_session::io_service()            //获得io_service
{ return m_socket.get_io_service();}
```

TCP 的打开和关闭

tcp_session 的关闭代码与前面相比没有变化：

```
void tcp_session::close()
{
  boost::system::error_code ignored_ec;                     //错误代码,不使用
  m_socket.shutdown(ip::tcp::socket::shutdown_both, ignored_ec);
  m_socket.close(ignored_ec);
}
```

在打开 TCP 连接的时候 tcp_session 需要创建一个 tcp_message 对象，然后把消息对象传递给启动异步读操作：

```
void tcp_session::start()
{
  tcp_request_ptr req = create_request();                   //创建消息对象
```

```
    read(req);                                          //启动异步读
}
```

create_request()使用内存池来创建 tcp_message 对象，注意代码中 bind 的使用方式—我们必须用 boost.ref 来包装内存池对象，因为它是不允许拷贝的：

```
tcp_request_ptr tcp_session::create_request()        //创建消息对象
{
    return m_msg_pool.construct(                       /内存池创建消息对象
        shared_from_this(),                            //传递 tcp_session 共享指针
        bind(&object_pool_type::destroy, ref(m_msg_pool), _1));//销毁器
}
```

TCP 读操作

因为现在的消息是由两部分组成的（消息头+消息体），所以 tcp_session 的读操作就需要变为三步：用 read()启动异步读，设置消息头的处理函数，然后在 handle_read_head()里处理接收到的消息头；如果消息头结构正确，那么再异步读消息体，最后在 handle_read_msg()里把收到的消息加入 job_queue，再调用 read()启动新的异步读：

```
void tcp_session::read( tcp_request_ptr req )            //启动异步读
{
    async_read(m_socket,                                 //使用自由函数 async_read
        buffer(req->head_data().data(),                  //使用消息里的消息头数据
               req->head_data().size()),
        m_strand.wrap(                                   //使用strand包装回处理函数
            bind(&tcp_session::handle_read_head, this, //绑定函数
            placeholders::error, placeholders::bytes_transferred,
            req))
        );
}

void tcp_session::handle_read_head( const system::error_code& error,
                                    size_t bytes_transferred,
                                    tcp_request_ptr req)
{
    if (error || !req->check_head())                     //检查消息头是否正确
    {
        close();
```

```
        return;
    }

    async_read(m_socket,                        //自由函数 async_read
        buffer(req->msg_data().data(),          //使用消息里的消息体数据
               req->get_head()->size),
        m_strand.wrap(                          // strand 包装处理函数
            bind(&tcp_session::handle_read_msg, this, //绑定函数
            placeholders::error, placeholders::bytes_transferred,
            req))
        );
}

void tcp_session::handle_read_msg( const system::error_code& error,
                                   size_t bytes_transferred,
                                   tcp_request_ptr req )
{
    if (error || !req->check_msg_crc())         //检查消息体是否正确
    {
        close();
        return;
    }

    m_queue.push(req);                          //把收到的消息加入到 job_queue

    start();                                    //启动新的异步读
}
```

TCP 读操作虽然复杂了些，但原理还是不变的，我们只需要以一个异步操作开始，然后“链式”编写处理函数，直至读完一个完整的消息。

注意在代码中我们使用 io_service::strand 的 wrap() 成员函数包装了 bind 处理函数，使操作可以安全地用于多线程环境，所以可以直接使用 this 指针，不必调用 shared_from_this()（当然使用它会更加安全）。

TCP 写操作

写操作与读操作类似，同样也是分三步走：

```
void tcp_session::write( tcp_response_ptr resp ) //启动异步写
{
```

```
  async_write(m_socket,                    //自由函数 async_write
      buffer(resp->head_data().data(),     //使用消息头数据
             resp->head_data().size()),
      m_strand.wrap(                       //strand 包装处理函数
          bind(&tcp_session::handle_write_head, this,//绑定函数
              placeholders::error, placeholders::bytes_transferred,
              resp))
      );
}

void tcp_session::handle_write_head( const system::error_code& error,
                                     size_t bytes_transferred,
                                     tcp_response_ptr resp )
{
  if (error ||                          //错误处理
      bytes_transferred != resp->head_data().size())
  {
      close();
      return;
  }

  async_write(m_socket,                 //自由函数 async_write
      buffer(resp->msg_data().data(),//使用消息体数据
             resp->get_head()->size),
          m_strand.wrap(                //strand 包装处理函数
              bind(&tcp_session::handle_write_msg, this, //绑定函数
              placeholders::error, placeholders::bytes_transferred,
              resp))
      );
}

void tcp_session::handle_write_msg( const system::error_code& error,
                                    size_t bytes_transferred,
                                    tcp_response_ptr resp )
{
  if (error ||                          //错误处理
      bytes_transferred != resp->get_head()->size)
  {
      close();
      return;
```

```
    }

    resp->destory();                              //完成消息的发送，销毁消息
}
```

注意在消息发送完毕后（handle_write_msg）我们调用了 tcp_message 的 destory()函数，因为这时候消息已经完成了它的使命，所以就应该及时归还给内存池。

12.4.4 tcp_server

tcp_server 与 12.2 小节的 tcp_server 类似，同样是非常简单，监听端口一旦有连接事件发生就创建一个 tcp_session。

tcp_server 需包含的头文件如下：

```
// tcp_server.hpp [2011 Aug chrono]
#include "tcp_session.h"
#include "io_service_pool.hpp"
#include <boost/bind.hpp>
#include <boost/functional/factory.hpp>
```

tcp_server 的成员变量多了 job_queue 的引用：

```
class tcp_server
{
public:
    typedef tcp_session::ios_type          ios_type;    //内部类型定义
    typedef boost::asio::ip::tcp::acceptor acceptor_type;
    typedef boost::asio::ip::tcp           tcp_type;
    typedef tcp_session::queue_type        queue_type;

private:
    io_service_pool& m_ios_pool;                      //并发线程池
    acceptor_type    m_acceptor;                      //接收器
    queue_type&      m_queue;                         //工作队列
```

这里我们为 tcp_server 编写两个构造函数，内部的 io_service_pool 是一个引用，既可以内部创建也可以外部传入：

```
public:
    tcp_server(unsigned short port, queue_type& q, int n = 4):
```

```
        m_ios_pool(*factory<io_service_pool*>()(n)),    //在堆上新建一个对象
        m_queue(q),                                     //初始化 job_queue
        m_acceptor(m_ios_pool.get(),                    //初始化接收器
            tcp_type::endpoint(tcp_type::v4(), port))
    {
        start_accept();                                 //开始监听端口
    }

    tcp_server(io_service_pool & ios, unsigned short port, queue_type& q):
        m_ios_pool(ios), m_queue(q),                    //初始化并发池和 job_queue
        m_acceptor(m_ios_pool.get(),                    //初始化接收器
            tcp_type::endpoint(tcp_type::v4(), port))
    {
        start_accept();                                 //开始监听端口
    };
```

start_accept()仍然是创建一个 tcp_session 的共享指针，然后调用接收器启动异步监听：

```
  private:
    void start_accept()                                 //启动端口监听，异步接受连接
    {
        tcp_session_ptr session =                       //创建 tcp_session
            factory<tcp_session_ptr>()(m_ios_pool.get(), m_queue);

        m_acceptor.async_accept( session->socket(),     //启动异步接受
            bind(&tcp_server::accept_handler, this,     //绑定处理函数
                boost::asio:: placeholders::error, session));
    }

    void accept_handler(const boost::system::error_code& error,
        tcp_session_ptr session)
    {
        start_accept();                                 //启动一个新的异步接受操作

        if (error)                                      //错误处理
        {
            session->close();
            return;
        }
```

```
        session->start();                              //启动 TCP 连接开始读数据
    }
```

最后是 tcp_server 的 start() 和 run() 操作：

```
    void start()
    {   m_ios_pool.start();}                           //非阻塞

    void run()
    {   m_ios_pool.run();}                             //阻塞
};                                                     //tcp_server 定义结束
```

12.4.5　实现 echo 协议

新的 TCP 服务器程序可以用来实现 echo 协议（其他的 discard、datetime 等协议读者可自行尝试实现）。

服务器端

服务器端的程序除了必备的 tcp_server 外我们还必须提供一个 job_queue 和相应的 worker 来存储和处理消息，代码很简单：

```
int main()
{
    cout << "echo server" << endl;                    //调试信息

    typedef tcp_server::queue_type queue_type; //消息队列定义
    queue_type msg_q;                                  //消息队列

    worker<queue_type> w(msg_q, check_msg);            //工作线程，使用回调函数
    w.start();                                         //启动工作线程

    tcp_server svr(6688, msg_q, 1);                    //服务器，工作在 6688 端口
    svr.run();                                         //启动服务器，开始处理 TCP 请求
}
```

工作线程 w 使用了 check_msg() 函数作为回调，它的功能很简单，显示收到的消息并原样转发：

```
bool check_msg(tcp_request_ptr& p)
{
  cout << p->get_head()->size << endl;                //显示消息长度

  p->msg_data()[p->get_head()->size] = 0;             //字符串末尾置 0
  cout << p->msg_data().data() << endl;               //显示收到的消息

  p->get_session()->write(p);                         //原样发送回客户端
  return true;
}
```

客户端

客户端程序中我们不能使用 tcp_session，因为它是服务器专用的类，不过直接用 boost::asio 的 socket 来实现也很容易：

```
int main()
{
  cout << "echo client" << endl;                      //调试信息

  io_service ios;

  ip::tcp::socket sock(ios);
  ip::tcp::endpoint ep(
      ip::address::from_string("127.0.0.1"), 6688);   //本地地址

  sock.connect(ep);                                   //连接到 echo 服务器

  string str("hello world");                          //待发送的消息

  msg_head head;                                      //初始化消息头结构
  head.type = 0x101;
  head.size = static_cast<uint32_t>(str.size());      //消息体大小
  head.chksum = std::for_each(str.begin(), str.end(),
      crc_32_type())();           //使用 for_each 算法配合 crc 对象计算 crc 校验

  sock.write_some(buffer(&head, sizeof(head)));       //发送消息头
  sock.write_some(buffer(str));                       //发送消息体

  sock.read_some(buffer(&head, sizeof(head)));        //接收消息头
  cout << head.size << endl;      //输出消息体大小，暂忽略 crc 验证
```

```
    vector<char> v(100, 0);
    sock.read_some(buffer(v));      //接收消息体
    cout << &v[0] << endl;          //输出消息体
}
```

12.4.6 实现聊天室

本小节我们来实现一个简单的聊天室程序，它包括服务器和客户端两部分，比 echo 协议要复杂一些。

聊天协议

首先我们定义一个非常简单的聊天协议，需要用到 msg_head 里的 type 字段：

- type==0：登录聊天室，消息体是用户名，暂无口令等验证机制；
- type==1：聊天的消息，消息体是用户实际的聊天内容；
- type==2：退出聊天室，消息体是用户名。

这三个类型整数可以用一个 enum 来描述。

```
enum {CHAT_JOIN, CHAT_MSG, CHAT_LEAVE};
```

服务器端

上一小节实现 echo 协议时我们直接在 main 函数里定义了 job_queue 和 worker，这种方式只适合很简单的服务程序，要实现聊天室最好把这些功能都封装到一个类里。

我们把这个类命名为 chat_server，它包括与消息相关的所有操作，包含的头文件如下：

```
//chat_server.h [2011 Aug chrono]
#include "tcp_message.h"
#include "job_queue.hpp"
#include "worker.hpp"
#include "safe_map.hpp"
```

chat_server 的定义如下：

```
class chat_server
{
public:
```

```
    typedef job_queue<tcp_request_ptr> req_queue_type;     //接受消息队列
    typedef worker<req_queue_type> req_worker_type;        //处理已收到的消息

    typedef job_queue<tcp_response_ptr> resp_queue_type;   //发送消息队列
    typedef worker<resp_queue_type> resp_worker_type;      //处理待发送的消息

    typedef safe_map<std::string, tcp_session_ptr> map_type;//管理连接

private:
    req_queue_type m_recvq;                        //接收消息的队列
    resp_queue_type m_sendq;                       //待发送的消息队列

    req_worker_type m_req_worker;                  //处理已接收消息的工作线程
    resp_worker_type m_resp_worker;                //处理待发送的消息的工作线程

    map_type m_sessions;                           //使用散列表管理所有 TCP 连接

public:

    chat_server();                                 //构造函数
    req_queue_type& recv_queue();                  //接收消息队列的访问函数
    void start();                                  //启动工作线程

private:
    bool process_msg(tcp_request_ptr& req);        //工作线程使用的回调函数
    bool send_msg(tcp_response_ptr& resp);         //工作线程使用的回调函数

    void register_user(tcp_request_ptr& req);      //注册用户
    void unregister_user(tcp_request_ptr& req);    //注销用户
    void deliver_msg(tcp_request_ptr& req);        //向用户发送消息
    void map_call_back(map_type::reference x,      //用于散列表的回调函数
        tcp_request_ptr& req);
};      //chat_server 定义结束
```

chat_server 里定义了两个 job_queue 和对应的 worker，分别用于接收从 tcp_server 获得的消息和准备向客户端发送的消息。为了管理聊天室里的用户，我们还使用 12.3.4 小节的 safe_map 来保存用户名与 tcp_session_ptr 的映射关系。

服务器端实现

chat_server 构造函数的唯一工作就是初始化两个 worker，使用 bind 绑定回调函数：

```
chat_server::chat_server():                          //构造函数
  m_req_worker(m_recvq,                              //处理接收到的消息
      boost::bind(&chat_server::process_msg, this, _1)),
  m_resp_worker(m_sendq,                             //处理待发送的消息
      boost::bind(&chat_server::send_msg, this, _1))
{      }
```

m_resp_worker 的回调函数 send_msg()非常简单，只需要从待发送队列中取出消息，然后用 tcp_message 里的 tcp_session_ptr 把它发送出去就可以了[①]：

```
bool chat_server::send_msg(tcp_response_ptr& resp)   //发送消息的回调函数
{
  resp->set_msg_crc();                               //设置好消息体的 crc 值
  resp->get_session()->write(resp);                  //发送消息
  return true;
}
```

process_msg()是 chat_server 的核心功能函数，它处理不同类型的消息，并把处理后的结果放置到待发送队列。为了简化消息类型的处理，它再调用三个子函数[②]：

```
bool chat_server:: process_msg(tcp_request_ptr& req) //处理接收到的消息
{
  register_user(req);                                //处理注册消息
  unregister_user(req);                              //处理注销消息
  deliver_msg(req);                                  //处理聊天消息

  return true;
}
```

用户的注册和注销我们做得很简单，不使用数据库，也不验证用户，用户的注册/注销操作总会成功，但并不向客户端返回确认消息。当用户注册时把名字和 TCP 连接加入散列表，注销时把他从散列表里删除：

```
void chat_server::register_user( tcp_request_ptr& req )  //注册用户
{
  msg_head* hp = req->get_head();                        //获得消息头结果
  if (hp->type != CHAT_JOIN)                             //判断是否是注册消息
  {   return;}
```

① 出于书写方便的原因，这里的代码都没有检查指针是否有效，请读者注意。

② 我们也可以把 process_msg()函数变成一个消息处理类 msg_processor，它可以独立演化。

```
    string name(req->msg_data().data(), hp->size); //得到用户名
    BOOST_ASSERT(!m_sessions.find(name));          //断言未注册到聊天室
    cout << "register " << name << endl;           //调试信息

    m_sessions.insert(name,req->get_session());    //加入散列表
  }

  void chat_server::unregister_user( tcp_request_ptr& req )//注销用户
  {
    msg_head* hp = req->get_head();                //获得消息头结果
    if (hp->type != CHAT_LEAVE)                    //判断是否是注销消息
    {   return;}

    string name(req->msg_data().data(), hp->size); //得到用户名
    BOOST_ASSERT(m_sessions.find(name));           //断言已经注册到聊天室
    cout << "unregister " << name << endl;         //调试信息

    m_sessions.erase(name);                        //从散列表中删除用户
  }
```

deliver_msg()调用 safe_map 的 for_each()成员函数，向聊天室里的每一位用户发送消息：

```
  void chat_server::deliver_msg( tcp_request_ptr& req )
  {
    msg_head* hp = req->get_head();        //获得消息头结果
    if (hp->type != CHAT_MSG)              //判断是否是聊天消息
    {   return;}

    cout << "deliver msg" << endl;         //调试信息
    m_sessions.for_each(                   //调用 safe_map 的 for_each()成员函数
        bind(&chat_server::map_call_back, this, _1, req));
  }
```

map_call_back()创建一个新的 tcp_message（tcp_response），把聊天信息拷贝到消息体，然后从散列表中获得其他用户的 TCP 链接，填充完整消息结构后投递到待发送消息队列。注意在创建消息的时候我们没有使用内存池，而是直接用 factory<>在堆上创建：

```
void chat_server::map_call_back( map_type::reference x,
                                 tcp_request_ptr& req)
{
  tcp_session_ptr sp = x.second;              //获得聊天室其他用户的 TCP 连接
  if (!sp->socket().is_open())
  {   return;}

  tcp_response_ptr resp = factory<tcp_response_ptr>()(sp);//创建消息
  resp->get_head()->size = req->get_head()->size;    //设置消息的长度
  std::copy(req->msg_data().begin(),                 //拷贝聊天消息
      req->msg_data().begin() + req->get_head()->size,
      resp->msg_data().begin());

  m_sendq.push(resp);                                //加入到待发送消息队列
}
```

最后是 chat_server 的两个辅助函数：

```
void chat_server::start()                            //启动工作线程
{
  m_req_worker.start();                              //启动接收消息的线程
  m_resp_worker.start();                             //启动发送消息的线程
}

chat_server::req_queue_type& chat_server::recv_queue() //访问函数
{ return m_recvq;}                                   //返回接收消息队列
```

服务器端主程序

聊天室的 main() 函数很简单，因为主要工作都已经被 tcp_server 和 chat_server 做完了，它的工作就是把这些组装起来并启动而已：

```
int main()
{
  chat_server cs;                                    //聊天服务器
  cs.start();                                        //启动聊天室工作线程

  tcp_server srv(6688, cs.recv_queue(), 1);          //TCP 服务器
  srv.run();                                         //启动 TCP 服务器
};
```

客户端

客户端程序中我们不能使用 tcp_session，因为它是服务器专用的类，不过直接用 boost::asio 的 socket 来实现也很容易，这里我们把客户端的功能封装到一个 chat_client 类里：

```
//chat_client.hpp [2011 Aug chrono]
class chat_client                                    //聊天室客户端类
{
public:
  typedef boost::asio::ip::tcp::socket socket_type;  //类型定义
  typedef boost::asio::io_service ios_type;
  typedef boost::asio::ip::tcp tcp_type;

private:
  ios_type &m_ios;
  socket_type m_socket;

  msg_head m_read_head;                              //消息头结构
  boost::array<char, 1024> m_read_data;              //接受聊天消息
  std::deque<std::string> m_send_msgs;               //暂存发送的消息
```

chat_client 的构造函数接受 io_service 和 endpoint，并异步连接聊天服务器：

```
public:
  chat_client(ios_type& ios, tcp_type::endpoint ep)://构造函数
      m_ios(ios), m_socket(ios)                      //初始化成员变量
  {   start_connect(ep);}                            //异步连接服务器
```

start_connect()在连接成功后启动异步读操作，接收服务器发来的聊天信息：

```
private:
  void start_connect(tcp_type::endpoint ep)                 //异步连接服务器
  {
      m_socket.async_connect(ep,                            //异步连接
          boost::bind(&chat_client::handle_connect, this,   //设置处理函数
          boost::asio::placeholders::error));
  }

  void handle_connect(const boost::system::error_code& error)
  {   read();}                                              //启动异步读
```

```
    void read()
    {
        boost::asio::async_read(m_socket,              //异步读消息头
            boost::asio::buffer(&m_read_head, sizeof(m_read_head)),
            boost::bind(&chat_client::handle_read_head, this,
                boost::asio::placeholders::error,
                boost::asio::placeholders::bytes_transferred));
    }

    void handle_read_head(const boost::system::error_code& error,
        size_t bytes_transferred)
    {
        if (error)
        {   return;}

        boost::asio::async_read(m_socket,              //异步读消息体
            boost::asio::buffer(&m_read_data[0], m_read_head.size),
            boost::bind(&chat_client::handle_read_msg, this,
            boost::asio::placeholders::error,
            boost::asio::placeholders::bytes_transferred));
    }

    void handle_read_msg(const boost::system::error_code& error,
        size_t bytes_transferred)
    {
        if (error)
        {   return; }

        cout.write(&m_read_data[0], m_read_head.size); //读到消息后输出
        cout << endl;

        read();                                        //启动新的异步读操作
    }
```

chat_client 使用 send() 函数发送聊天消息，这里我们使用简单的同步操作：

```
public:
    void send(const std::string& str)                  //发送聊天消息
    {
        m_send_msgs.push_back(str);                    //把消息保存到 deque 中
```

```
        write();                    //调用内部的发送消息函数
    }
private:
    void write()                    //内部的发送消息函数
    {
        msg_head head;
        const std::string& str = m_send_msgs.front();

        head.type = 1;              //填充消息头结构
        head.size = static_cast<uint32_t>(str.size());
        head.chksum = std::for_each(str.begin(), str.end(),
            crc_32_type())();

        m_socket.write_some(buffer(&head, sizeof(head)));  //发送消息头
        m_socket.write_some(buffer(str));                  //发送消息体

        m_send_msgs.pop_front();                           //消息发送完毕
    }
```

最后是简单的登录、退出和关闭函数：

```
public:
    void login(const std::string& name)                    //用户登录
    {
        msg_head head;

        head.type = 0;                                     //填充消息头结构
        head.size = static_cast<uint32_t>(name.size());
        head.chksum = std::for_each(name.begin(), name.end(),
            crc_32_type())();

        m_socket.write_some(buffer(&head, sizeof(head)));  //发送消息头
        m_socket.write_some(buffer(name));                 //发送消息体
    }

    void logout(const std::string& name)                   //用户退出
    {
        msg_head head;

        head.type = 2;                                     //填充消息头结构
        head.size = static_cast<uint32_t>(name.size());
```

```
        head.chksum = std::for_each(name.begin(), name.end(),
            crc_32_type())();

        m_socket.write_some(buffer(&head, sizeof(head)));  //发送消息头
        m_socket.write_some(buffer(name));                 //发送消息体
    }

    void close()                                           //关闭 TCP 连接
    {   m_socket.close();}
};     //chat_client 定义结束
```

客户端主程序

聊天客户端的代码要稍微复杂些，因为它需要处理用户的输入。我们使用 std::getline()函数来从标准输入流 cin 获取用户输入的聊天信息，直至输入“quit”退出：

```
#include "chat_client.hpp"

int main()
{
  cout << "chat client" << endl; //调试信息

  io_service ios;
  ip::tcp::endpoint ep(
      ip::address::from_string("127.0.0.1"), 6688);

  chat_client client(ios, ep);    //创建聊天客户端

  boost::thread t(                //用线程启动 asio，在后台运行消息的收发
      boost::bind(&boost::asio::io_service::run, &ios));

  string name;
  cout << "please input name:";
  getline(cin, name);             //获得用户名
  client.login(name);             //注册到聊天服务器

  string str;
  while(getline(cin, str))        //获取用户的输入
  {
      if (str == "quit")          //用户输入“quit”则退出
      {   break;}
```

```
        str = name + " say: " + str;     //拼一个聊天信息
        client.send(str);                //发送到聊天服务器
    }

    client.logout(name);                 //退出聊天服务器
}
```

现在我们就完成了整套聊天室程序，可以启动一个服务器和多个客户端来测试它们是否工作正常，在任何一个客户端输入的消息都会立刻出现在其他客户端的界面上。

12.5 总结

本章是“理论联系实际”的一章，开发实现了两个可用的 TCP 服务器，研究了如何具体使用 Boost 的各种组件。

首先利用 Boost 库开发了一些基本工具，包括标准整数、并发线程池、阻塞队列、工作线程以及线程安全的容器和单件，这些工具在任何项目中都是基础，基于它们才能搭建出稳定牢固的程序。基本工具有很多，本章只实现了很少的一部分，更多的需要读者结合自己的实际需要去开发。

多线程编程和并发编程是热门的议题，Boost 中的 thread 和 asio 库恰好覆盖了这两个领域，而且经常搭配 function 和 bind 一起应用，它们功能完备强大且使用方便，值得去仔细研究学习。

有了基本工具之后，我们基于 asio 库开发了两个结构略有差异的 TCP 服务器程序，并实现了多个网络协议。它们的核心是使用了前摄器模式的 io_service 对象，可以异步地发起 TCP 操作，再适当地配合多线程就可以更好地利用多核 CPU 的能力。第一个 TCP 服务器操作的是无格式的字节流，使用“智能接口”的回调函数处理数据；而第二个 TCP 服务器操作的是自定义格式的字节流，使用工作队列+工作线程来处理数据，两者各有优缺点。当然我们也可以把这两者糅合起来，在回调函数中解析有格式的字节流，然后再用多线程的工作队列处理数据，这样会更灵活方便，也更能够充分利用 CPU 的计算能力。

本章的代码较多，而且较偏重于工程实践，使用的 Boost 组件简单列举如下：

- assert ：运行时断言，较标准的 assert 宏功能更多；
- integer ：定义精确的整数类型；

- smart_ptr ：主要使用的是 shared_ptr，安全地管理指针；
- pool ：内存池，可以高效地分配内存；
- factory ：消除直接使用 new 关键字；
- date_time ：日期时间库；
- circular_buffer：环形的循环容器；
- unordered ：高效的散列容器；
- ptr_container ：使用指针容器管理不可复制的对象；
- crc ：计算 crc 校验码；
- function ："智能函数指针"，可以容纳任意可调用物；
- bind ：功能强大的函数绑定器，现代 C++函数式编程的重要武器；
- thread ：完备易用的多线程序库；
- asio ：使用前摄器模式的并发库，并不局限于网络编程。

这些组件的使用原则都可以在第 13 章找到依据。当然，Boost 中还有许多其他常用的并且功能强大的组件本章并没有涉及，如 filesystem（文件系统）、xpressive（正则表达式）、signals2（观察者模式），还需读者在书外多多自行体验。

第13章

Effective Boost

本章是全书的最后一章，在这里我们将不再探究 Boost 程序库的细节和用法，而是讨论与它相关而又在它之外的东西——如何使用 Boost 编写清晰、优雅、高效、灵活和易维护的 C++代码。

C++是一门伟大的语言，几乎能够胜任任何工作，但却对普通程序员不太友好，存在着太多的“未定义行为”的陷阱（而其他编程语言，如 Java 和 C#，则好的多），如同某些小说中的奇门异宝，威力虽大但使用不当也容易反噬自身。高效地使用 C++的优点，同时避免 C++的缺点，是 C++程序员永恒的功课。

已经有许多专家学者论述了如何正确高效地使用 C++（如推荐书目[6]和[8]），其中的许多经典条款也被大众所熟知，如多态基类应使用虚析构函数、避免过长的函数（笔者曾经见过一个上千行的函数）、避免 new 动态分配内存（这通常是 80%C++代码错误的根源）等。相信读者也已经阅读过这些经典著作，笔者也不必就此多费口舌，仅把注意力集中在 STL 和 Boost 的使用方面，限于个人开发经验不一定完全正确（而且有时候原则是可以违反的），论述也难免简陋，愿与读者共同探讨。

13.1 基本原则

熟悉并使用 STL/Boost 编程范式

C++发明至今已经三十多年了，从最早的简单面向对象逐渐发展成为包含泛型、函数式、模板元等许多范式的复杂混合体，其中的每一个编程范式都可以自成体系，在开发过程中打出一片天地。

二十年前，面向过程、基于对象是 C++编程的主流范式；十年前，主流范式变成了面向

对象+设计模式，而现在，C++编程的主流范式则有“返璞归真”的趋势，过度使用虚函数的庞大类继承体系逐渐被摈弃，而使用泛型、函数式等新范式开发精致的小类并给予良好的组合成为了大方向[①]。

STL 和 Boost 充分实践了现代 C++编程方法，不使用复杂的继承体系（少数例外，如 iostreams），特别是 Boost，它使用泛型编程、模板元编程和编译期的静态多态构建了功能完善的组件，代码规范精炼，是我们学习的极好范例。

STL/Boost 编程范式通常会要求我们编写带有模板参数的泛型代码，使用 typedef 进行类型计算，同时因为编译器模板实例化的原因，功能实现代码通常都写在 hpp 头文件里，最后由少量 cpp 完成功能的组装。

尽量理解 Boost 的工作原理

Boost 程序库庞大复杂，内部又十分精致细腻，由于提供了丰富的自说明文档，只要多花心思和时间，了解功能组件的接口和用法应该不是什么难事。

但在掌握了 Boost 组件的基本用法之后，我们应该再多用些时间去查看它们的实现代码：一是学习库作者的设计思想，学习这些顶尖大师的经验，二是了解内部实现机制和运行原理，从而能够洞悉其优缺点——很多细节并不是都在文档中有描述的。这样在使用时自然就能够避免一些性能不佳的用法，提高 Boost 组件的运行效率，达到“知其然更知其所以然”的境界。

boost.optional 是一个学习 Boost 库及模板元编程的好例子，它足够小却又并不简单，为了支持容纳 T 和 T&两种模板类型使用了一些 type_traits 和 mpl 里的元函数，并用一个 detail::reference_content 类型来保存值，这些巧妙的手法都值得我们认真揣摩。

理解 Boost 的工作原理必备的技能是模板元编程，相关知识可见第 1 章和第 11 章。

学会使用对象包装 C++原始语言概念

基于操作符重载、面向对象和泛型的强大威力，在 STL 和 Boost 中许多“原始”的 C++操作概念都有了对象版本，是更“智能”（smart）的操作。这些“智能操作”的形式和用法都与原始操作非常类似，因而很容易学习和使用，而且它们还具有许多原始操作所不具备的“智能”特性。使用这些智能操作可以更好地编写出健壮稳固的代码，也更利于代码的长期维护，所以我们应当尽量避免使用原始的 C++语言要素（如 new、delete、函数指针），而是

① 当然，这并不是说我们要完全不使用面向过程和面向对象，而是说应该逐渐少使用它们，逐渐转向泛型等新范式。

使用它们对应的“智能”对象版本，包括：

- 智能创建 ：factory，可取代操作符 new；
- 智能删除 ：checked_delete，可取代操作符 delete；
- 智能引用 ：reference_wrapper，可取代原始引用；
- 智能初始化 ：value_initialized，可取代原始初始化；
- 智能指针 ：auto_ptr/smart_ptr，可取代原始指针；
- 智能函数指针：function，可取代原始函数指针；
- 智能函数 ：各种函数对象以及 bind 和 lambda 表达式；
- 智能结构体 ：tuple，可取代简单组合数据的 struct[①]；
- 智能参数 ：call_traits，可取代简单的函数接口参数声明；
- 智能数组 ：boost::array/std::vector，可代替原生静态数组和动态数组。

尽量使用 exception 代替错误返回码

异常已经成为了 C++标准的一个不可或缺的组成部分，不论我们编写的代码是否使用异常，C++都会使用异常处理机制，因此我们不必担心异常处理会带来额外的运行开销[②]。相反，我们应该充分利用异常处理机制，简化对错误（即异常）的处理，摈弃 C 风格的错误返回码的方式——代码的主处理流程会只有正常的处理逻辑，不再需要麻烦的 if 错误检查语句，错误处理都会集中在某个特定位置的 catch 块中。

C++98 标准中提供了标准异常类 std::exception，但功能还比较有限，要把它用在自己的工程中还需要多做许多工作。boost.exception 库在标准异常的基础上进行了大幅度的强化，可以向异常对象中添加任意的信息，轻松构建出任意复杂的异常体系，增强了异常的表达能力，令异常处理变得更加简单。

使用异常时还需要避免过度使用 try-catch 块，避免多层嵌套，尽量多使用简洁的 function-try 形式。

---

① 在 C 中 struct 关键字仅起到对数据打包组合的作用，但在 C++中 struct 基本上是 class 的同义词，在这里的意思是 tuple 可取代 C 用法的 struct。

② 在 Java 和 C#等编程语言中异常已经得到了广泛且大量的使用和验证，足以证明异常机制的可靠性和效率。

恰当地使用新式转型操作符

转型操作是 C 语言的“遗产”，通常我们应该尽量少用强制转型，因为这通常表明我们设计的接口有问题，真正好的代码应该是无须转型就可以使用的，不会发生编译警告。

但有的时候——通常是我们在与 C API 打交道时——我们又必须用强制转型，在这里建议读者透彻地理解标准库的四个新式转型操作符（参见 2.6 小节）和 Boost 提供的几个转型工具，彻底消灭编译时的转型警告。虽然这样可能会让代码显得有些冗长繁杂，但它会增强代码的可靠性，避免了代码的含糊语义。新式转型操作符还有一个额外的好处，可以很容易地使用诸如 grep 等文本工具处理。

例如，如果把一个 string 类型的数据转换为一个 byte 数组，代码为：

```
void sha1(byte_t *indata, ulong_t len);  //计算 sha1 摘要的 C API

string str("abc");
sha1(reinterpret_cast<byte_t*>(const_cast<char*>(str.data())),
     numeric_cast<ulong_t>(str.size()));
```

其中的 numeric_cast<ulong>通常也可以使用 static_cast<ulong_t>来代替（如果不想使用 numeric 组件的话），但这可能会带来一点点安全隐患。

使用 test 库进行测试驱动开发

测试是软件开发过程中一项非常重要的工作，极限编程/敏捷开发都强调测试驱动开发，要求测试代码要先于功能代码开发，功能代码开发的目的就是保证测试代码通过。

虽然不是所有工程都适合使用极限编程，但对测试的重视无论如何都是非常必要的。在 boost.test 库出现之前，为 C++代码编写单元测试是一件颇为麻烦的工作，需要做大量与测试本身无关的外围工作。boost.test 库构建了一套完整的测试框架，我们只需要使用几个简单的宏就可以创建整套的测试夹具、测试用例和测试套件，大大简化了编写测试代码的工作，使之成为了一种“享受”，令人简直无法抗拒写单元测试的诱惑。

多使用 Boost 小工具来增加代码的健壮性

Boost 程序库包罗万象，其中的组件大的分类就达到上百个，而每个组件里面还有更多更小的组成部分，让人眼花缭乱目不暇接。很多程序员常常会把注意力集中在 Boost 库中的那些重量级组件上（如 thread、asio、xpressive、signals、bind），而不自觉地忽略了那些功能小而简单的组件，认为这些小工具无关大局，用不用无所谓—个人认为这种想

法是相当错误的，正所谓“勿以善小而不为”。

所有的成功都是由无数的细节搭建起来的，编写程序也是一样，Boost 库中有许多非常实用的小工具，适当地使用它们可以很好地改善代码的可读性和健壮性，例如：

- typeof 可以简化变量的类型声明；
- foreach 可以很容易地遍历容器内所有元素；
- noncopyable 可以明确地定义一个不允许拷贝的类；
- integer 定义了标准整数；
- factory 封装了 new 操作符；
- array 封装了原始数组概念；
- tribool 提供三态布尔逻辑；
- utility 库包含数个有用的小工具；
- 还有更多……

学习并熟练使用这些小工具将会丰富我们的编程词汇，使我们的每一行代码都能够达到 Boost 源码那样的优雅程度。

13.2 内存管理

学会使用 Boost 管理内存

C++在可预见的一段时期内还不会有垃圾回收机制，手工管理内存还是程序员们必需的工作，这虽然给予了我们最大的灵活性，但也带来了许多的麻烦。

基本的内存管理方法有很多，可以使用 C 语言的 malloc/free 分配释放内存，也可以使用 new/delete 关键字分配释放内存，我们也可以重载 operator new/delete 来定制内存分配策略，但这些方式都比较“原始”。

Boost 使用 pool 库提供了一个内存池功能，它基于简单分隔存储思想，不一定是最好最高效的内存池实现，但足够快速，并且简单易用。我们可以使用它预先向系统申请大块的内存，然后在里面自行取用，避免了反复用 new/delete 向系统申请释放内存，可以极大地提高内存的使用效率。

智能指针和指针容器也是管理内存的有效手段，它们不仅能够管理 new 分配的内存，也可以管理内存池分配的内存（使用删除器或克隆分配器）。

使用智能指针代替原始指针

指针是 C/C++中一个重要的概念，但它用起来缺乏足够的安全性，会导致很多潜在的问题。智能指针是一个“伟大的发明”，它使用 RAII 很好地解决了内存泄漏的问题，同时又没有带来过多的性能损失，对于广大程序员来说是不可或缺的工具。

boost.smart_ptr 库提供了数个优秀的智能指针，特别是其中采用引用计数的 shared_ptr，几乎完全可以代替原始裸指针，应用范围非常广泛，我们应该尽量在自己的代码中使用。

C++98 标准中也提供了一个智能指针：auto_ptr，但它因为微妙的转移语义而饱受恶评，很多人都不推荐甚至不喜欢使用 auto_ptr，但在作者看来它却并非一无是处。虽然它存在一些缺陷，但毕竟是目前 C++标准中唯一可用的智能指针，我们只要适当地使用它同时避免误用就完全可以发挥它的真正价值。而且 auto_ptr 由于其独特的“转移”语义更是可以和指针容器完美搭配，使用 auto_ptr 动态创建对象再放入指针容器可以彻底避免内存泄漏。

尽量避免直接使用 new/delete

智能指针和 checked_delete 可以很好地消除 delete 关键字的使用，但 new 关键字的调用有时候还是不可避免。好在 Boost 也给我们提供了替代的方式，例如 make_shared() 和 factory<>，前者可以创建 shared_ptr，后者更可以创建任意的指针类型。这些 Boost 工具通过引入间接层封装了原始概念的使用，因而可以更好地对内存进行管理。

13.3 容器、迭代器和算法

了解现有的容器

STL 和 Boost 都提供了大量的容器类型，种类繁多，要从中选择一个适合于自己应用的容器不是一件容易的事情，把这些容器分门别类进行整理划分有助于我们快速定位所需的容器：

- 按实现方式可分为侵入式容器和非侵入式容器，侵入式容器目前仅有 boost.intrusive，其他的容器都属于非侵入式容器；

- 按容纳元素的类型可分为值容器和指针容器，值容器容纳的是元素的拷贝，指针容器容纳的是指针；
- 按容器的数据结构可分为线性容器、树结构容器和散列容器，三者各有优缺点；
- 按元素的访问方式可分为序列容器和关联容器，而多索引容器 boost.multi_index 则结合了两者的特点；
- 此外还有编译期可容纳类型的 mpl 容器，用于模板元编程。

有了这些容器，我们就完全可以不使用 C/C++在底层语言级别提供的数组/动态数组，可以更高效地利用资源。最常用的容器是 std::vector、std::map、boost::array 和 boost::unordered_map，其他的容器则应该根据具体的应用场景取舍。

还需要注意的是，这些容器基本上都不保证线程安全，在并发环境中使用时如果必要需加锁。

恰当地使用指针容器

指针容器是 Boost 库对容器做出的一个重要扩展，它可以安全地容纳指针，消除了元素拷贝的代价，同时它具有与标准容器基本相同的接口，因而学习成本低上手容易。

与标准容器相比指针容器有很多优点，特别适用于那些需要管理大对象的场合，这时管理指针比管理对象的拷贝要高效许多。但使用它也不是没有代价的，由于指针的特殊性，它的基本概念要比标准容器复杂（如对空指针的处理），涉及到一些底层细节时如果不留心很可能会出错。

因此，恰当并且审慎地对待指针容器才是正确的用法。

编写新容器或迭代器应满足概念要求

虽然 STL 和 Boost 提供了足够多的容器，但有的时候我们还是需要编写新的容器或者迭代器来操纵特定的数据结构，这时不应仅仅以实现容器或迭代器的可用接口为目标，而是应该提高要求——新的容器或迭代器应该满足标准概念，这样它才能更好地与 STL 或 Boost 库的组件配合工作。

很多 Boost 组件都要求容器或迭代器满足标准概念，例如，如果容器的迭代器没有定义 iterator_category，那么就无法使用 boost.foreach。

对于容器来说，它应该具有一些必备的接口，如 size()、max_size()、begin()、end()等，同时还必须有大量的内部 traits 类型定义；对于迭代器来说则应该具有

operator*、operator->等接口，还要有 value_type、reference 等 traits 定义。容器或迭代器是否满足标准概念的要求可以使用概念检查库 concept_check 来验证。

多使用 boost.foreach 算法

循环是计算机程序的基本结构，遍历一个容器中的所有元素是我们经常要做的工作。最早的 for 是基于整数计数的循环（++i），在迭代器概念出现后又变成了迭代器的移动(++iter)，但始终是一成不变的“for(初始化;条件判断;前进)”的形式，非常的不优雅，语法要素太多很容易导致笔误。

标准库的 for_each 算法一定程度上解决了这个问题，它需要指定容器的两个端点，然后对其中的元素执行某个操作。但它也有缺点，为了使用 for_each 我们必须额外编写一个函数或函数对象，本来应该减少的代码编写工作反而增加了，这也令不少人不愿意使用 for_each 算法，而宁愿继续手写 for 循环。

boost.foreach 使用模板元编程技术彻底地解决了遍历容器的问题，只需要一个简单的宏就可以正序或逆序遍历容器里的所有元素，用法与 for 循环完全一样，这样的“语法糖”带来的好处是显而易见的，绝不只是少打几个字符那么简单。

当然 boost.foreach 现在还存在一些不足，但在 C++语言基本的遍历容器 for 功能出现之前，它是容器的最佳搭档。

13.4 其他

熟悉 Boost 库中已有的设计模式实现

设计模式是领域专家的经验结晶，给出了针对特定问题的最佳解决方案，自 1995 年推荐书目[2]出版以来，各式各样的模式大量涌现，许多常见的问题都可以找到对应的模式，极大地促进了软件业的发展。

需要强调的是，设计模式不单纯指面向对象的解决方案，不是单纯的类的嵌套与组合，它更是一种解决问题的思想和思路。虽然经典的 23 个模式都是基于面向对象的，而 STL 和 Boost 主要使用的是泛型，但这并不影响设计模式的使用。

Boost 库中的许多组件的设计和实现都深受设计模式的影响（例如代理模式就有 smart_ptr、optional、array 等，推荐书目[1]中对此有较好的总结，读者可参考），熟悉这些组件使用的设计模式不但可以加深对 Boost 的了解，而且也可以反过来加深对设计

模式的了解，可谓一举两得。

熟悉多线程领域的基本概念和模式

boost.thread 库提供了用于多线程编程的大量工具，方便易用，但仅有这些工具还是不够的，如果不明白如何正确地使用这些工具那么它们也无法发挥出应有的作用。

多线程开发有许多不同于单线程开发的新问题，要避免或者解决这些问题就需要透彻理解线程安全、可重入等概念，在编写代码时时刻考虑多线程并发的情况，避免使用全局变量、成员变量或静态变量，根据具体情况使用 boost.thread 提供的互斥量、条件变量和线程局部存储等功能。

多线程开发还有许多成熟的范式，如主动对象、future 等，熟悉这些范式有助于我们构建更稳固的多线程程序。

留意 Boost 与 TR1、C++11 实现的区别

Boost 被称为“C++‘准’标准库”，是 C++新标准的试验场和后备军，许多新组件都需要经过 Boost 的锤炼然后才能进入标准。tr1 中来自 Boost 库的就有 shared_ptr、function、bind、array、ref、type_traits，tr2 中则可能有 system、filesystem。

对于那些不是来自 Boost 的 tr1 组件，Boost 也提供了实现（如 unordered)，并且专门有一个 tr1 库与 C++ tr1 对应，使用 boost.tr1 我们可以毫无困难地在 Boost 实现与标准实现之间自由切换。此外 Boost 还用模板元编程技术模拟实现了 typeof、foreach、concept 等 C++11 中才有的新特性。

但在使用 boost.tr1 时需要注意，它并不是完全遵照 tr1 标准实现的，虽然大部分功能一致，但也有极少的部分存在差别，如果使用到这些差异细节就可能会遇到不可移植的问题。这其中特别值得一提的是 boost.ref 库，时至今日仍然没有提供 operator()。

谨慎使用 bind、正则表达式、模板元编程等功能强大且灵活的库

Boost 中有许多功能强大的组件，如 bind、bimap、xpressive、mpl 等，它们专注于某一特定领域，用法灵活复杂，能够解决许多实际的问题。例如 bind 就被应用于函数式编程，使用占位符创建出新的函数对象，是 function、signals2、thread、asio 等库的必备搭档。但这些组件功能强大的另一方面是用法的复杂，学习并掌握它们需要花费相当多的时间，并且过于灵活的用法也会产生副作用—代码有如“天书”，增加以后维护的成本。

“过犹不及”，谨慎地使用这些高级技术会让程序更易读—我们编写的代码决不能成为个

人编程水平的技术炫耀，代码写出来应该是给“人”看而不是给“计算机”看的。

让你周围的同事熟悉 Boost

作为“Effective Boost”的最后一条，这也许是最重要的。

软件开发绝不是一个人短期能够完成的事情，而是许多人协同的长期工作，包含了需求、设计、测试、维护等多个阶段。Boost 的确是强大的武器，但正所谓“独乐乐，众乐乐，孰乐？”，如果仅仅只有自己一个人掌握了 Boost 的用法，那么他在工作过程中就避免不了“自说自话”的窘境，写出的代码没有人能看懂，也就没有人能够为他做 code review，发现其中可能存在的 bug，他将不得不“孤军奋战”，一个人自己去研究解决 Boost 使用中遇到的各种问题，可谓“高处不胜寒”[①]。

知识的传递是一件快乐的事，好的东西更应该与大家一起分享，把 Boost、C++的最前沿开发技术在自己的周围传授可以达到“众人拾柴火焰高”的境界。在这个知识分享的过程中我们不会有任何的损失，相反，还有可能因为不同人的思维角度的不同产生“头脑风暴”的效果，迸发出更多的思维火花，进一步加深我们对 Boost 和 C++的理解，提高我们的编程水平——这也正是笔者编写本书的原始动力。

13.5 结束语

本章从笔者自身经历出发，简要阐述了一些在实际开发工作中使用 Boost 的经验，限于文笔只能表述出一鳞半爪，不能说是金科玉律，但至少是肺腑之言。

开发出高质量高性能的软件是每一个程序员的梦想，每一个人都有他自己的 Effective 开发习惯，但有一点肯定是共通的：一个 Effective 程序员必定是一个快乐的程序员（至少是快乐的 C++程序员），希望读者在阅读完本章乃至本书后都能够 Effective and Happy，高质量且快乐地度过生活中的每一天。

① 当然也不排除少数人就喜欢这种“独孤求败”、“唯我独尊”的感觉，但这样不合群的“技术专家”还是越少越好，希望读者没有遇到过。

附录 A

推荐书目

[1] 罗剑锋著，《Boost 程序库完全开发指南——深入 C++“准”标准库》，北京：电子工业出版社

“举贤不避亲”，位于推荐书目首位的是本书的“前传”，被我昵称为“Boost 黑皮书”。本书是国内的第一本详细介绍 Boost 程序库的技术书籍，基于 2010 年 2 月份发布的 Boost1.42 版，面向 Boost 初学者，内容详细丰富，可以放在手头随时查阅，是一本很好的 Boost 工具书。

[2] Erich Gamma 等著，李英军等译，《设计模式：可复用面向对象软件的基础》，北京：机械工业出版社

这本书永远位于作者推荐榜的最前列。软件开发历史上里程碑式的著作，设计模式的开山作品，内容权威经典，远非其他“白话”之类可比，是每一个精益求精的程序员都必须拥有的宝典和圣经，值得经常阅读以获取设计灵感。

[3] Niolai M.Josuttis 著，侯捷/孟岩译，《C++标准程序库——自修教程与参考手册》，武汉：华中科技大学出版社

这本书同样是作者强力推荐的 C++书籍，全面分析讲解 C++98 标准库（特别是 STL），无论是 C++初学者还是高手都能从中获益，C++程序员必备，厚达 800 余页的大部头组织的井井有条，查阅起来毫不费力。

[4] Bjarne Stroustrup 著，裘宗燕译，《C++语言的设计和演化》，北京：机械工业出版社

C++语言之父所著，介绍了 C++语言的发展历史、设计理念，同时也详细描述了 C++的各种语言特性，从中可以更深刻地理解 C++语言的内涵，虽然出版时间较早但仍不失为一本好的参考书。

[5] Stanley B Lippman 等著，潘爱民等译，《C++ Primer 第三版》，北京：中国电力出版社

虽然目前本书已经有了最新的第五版，但这本书仍然没有过时，对于 C++98 标准的讲解仍然十分全面透彻，如果在某个旧书摊遇到请不要错过。

[6] Scott Meyers 著，侯捷译，《Effective C++ 中文版 第 3 版》，北京：电子工业出版社

享誉世界的“Effective 系列”，提供了 55 个如何编写高效 C++代码的忠告。

[7] Martin Fowler 著，侯捷/熊节译，《重构：改善既有代码的设计》，北京：中国电力出版社

本书讲述了若干“代码坏味”和重构准则，有的甚至相当简单琐碎，但的确是很有价值的实用工程准则，了解这些准则有助于我们更好地改善代码质量。不过全书使用 Java 编写范例代码，对于我们这些 C++程序员来说阅读起来略微有些不便。

[8] Herb Sutter 等著，刘基诚译，《C++编程规范：101 条规则、准则和最佳实践》，北京：人民邮电出版社

本书由两位 C++大师联手撰写，类似《Effective C++》，把专家们数十年的经验和智慧凝结成 101 条简练的规范，熟悉并运用这些准则可以大大提高我们的 C++编码水平（Boost 的许多组件直接对应其中某些条款的解决方案，如第 25 条对应 call_traits、第 27 条对应 operators、第 95 条对应 noncopyable）。

附录 B

Boost 程序库组件索引

本文按字母顺序列出 Boost 1.47 版（2011 年 7 月发布）中所包含的所有组件，并附简要说明，对于 1.42 版之后的一些重要变化用脚注的形式给出，供读者参考。

有*标记的组件表示在推荐书目[1]中有详细阐述。

A

accumulators：是一个用于增量统计的库，也是一个用于增量计算的可扩展的累加器框架，可以看做是 std::accumulate 算法的扩展。它的作者是 Eric Niebler。

any*：可以容纳任意类型数据的容器，有了它，C++的强类型特性似乎失去了效力。它的作者是 Kevlin Henney。

array*：包装了 C++语言内建的数组，为其提供标准的 STL 容器接口，速度、性能上与原始数组相差无几。它的作者是 Nicolai Josuttis。

asio*：基于操作系统提供的异步机制采用前摄器设计模式（Proactor）实现了可移植的异步（或者同步）IO 操作。它主要关注于网络通信方面，封装 socket API 提供了一个现代 C++风格的网络编程接口。但 asio 的异步操作并不局限于网络编程，还支持串口读写、定时器、SSL 等功能。asio 自 1.35 版加入 Boost 后一直在持续更新，很有活力。它的作者是 Chris Kohlhoff[1]。

assign*：重载了赋值操作符、逗号操作符和括号操作符，可以用难以想象的简洁语法非常方便地对 STL 容器赋值或者初始化，在需要填入大量初值的地方用处很大。它的作者是

① 在 Boost 1.47 版中 asio 增加了对 Unix/Linux 信号的处理功能，使用很方便。

Thorsten Ottosen。

B

bimap*：扩展了标准库的映射型容器，提供双向映射的能力，功能强大，其接口被特意设计为符合 STL 规范，以减少学习的负担。它的作者是 Matias Capeletto。

bind*：是 C++98 标准库中函数适配器 bind1st/bind2nd 的泛化和增强，可以适配任意的可调用对象，包括函数指针、函数引用、成员函数指针和函数对象，远远地超越了 STL 中的函数绑定器。它的作者是 Peter Dimov。

C

call_traits：封装了可能是最好的传递参数给函数的方式，会自动推导出最高效的传递参数的类型，而且保证不会出现“引用的引用”这个非法的错误。它的作者是 John Maddock、Howard Hinnant 等。

chrono:它是一个与 date_time 类似的时间处理库，但侧重于对时间点、时间间隔（duration）而不是日期的处理。它的作者是 Howard Hinnant、Beman Dewes 和 Vicente J. Botet Escribá①。

circular_buffer*：实现了循环缓冲区的数据结构，支持标准的容器操作，但大小是固定的，当到达容器末尾时将自动循环利用容器另一端的空间。它的作者是 Jan Gaspar。

compatibility：主要用于 Boost 库作者，对于某些不符合标准的实现提供变通解决办法。它的作者是 Ralf Grosse-Kunstleve 和 Jens Maurer。

compressed_pair:提供一个与 std::pair 非常相似的模板类 compressed_pair，用法也完全相同，不同之处在于 compressed_pair 使用了空基类优化技术。它的作者是 John Maddock、Howard Hinnant 等人。

concept check：实现了 C++11 草案中被否决的概念检查功能，可以在编译期检查模板函数或模板类的模板类型参数是否符合某个概念。它的作者是 Jeremy Siek。

config*：主要用于 Boost 库作者，解决 Boost 库组件对各种编译器、平台的兼容性问题。它的作者是 Beman Dawes、Vesa Karvonen、John Maddock。

conversion：增强了 C++标准中的 static_cast<>、dynamic_cast<>等转型操作符。它的作者是 Dave Abrahams 和 Kevlin Henney。

---

① 很荣幸，Boost 中能够有一个与我同名的库。

crc*：实现了计算循环冗余码（CRC）的功能，它的作者是 Daryle Walker。

D

date_time*：是一个非常全面且灵活的日期时间库，基于我们日常使用的公历提供时间相关的各种所需功能。它的作者是 Jeff Garland。

dynamic_bitset*：类似标准库的 bitset，提供丰富的位运算，同时长度又是动态可变的。它的作者是 Jeremy Siek 和 Chuck Allison。

E

enable_if：允许模板函数或者模板类仅针对某些特定类型有效，即启用或禁用某些特化形式，依赖于 SFINAE。它的作者是 Jaakko Jarvi、Jeremiah Willcock 和 Andrew Lumsdaine。

exception*：针对标准库中异常类的缺陷进行了强化，提供 operator<<重载，可以向异常传入任意数据，有助于增加异常的信息和表达力。它的作者是 Emil Dotchevski。

F

filesystem*：是一个可移植的文件系统操作库，使用 POSIX 标准表示文件系统的路径，使 C++具有了类似脚本语言的功能，可以跨平台操作目录、文件，写出通用的脚本程序。它的作者是 Beman Dawes[①]。

flyweight：实现了享元（flyweight）设计模式，可以管理大量的小对象以节约内存的使用。它的作者是 Joaquín M López Munoz。

foreach*：使用宏提供新式的序列遍历，简便好用，不需要使用麻烦的迭代器，也不需要定义新的函数对象。它的作者是 Eric Niebler。

format*：实现了类似于 printf()的格式化对象，可以把参数格式化到一个字符串，而且是完全类型安全的。它的作者是 Samuel Krempp。

function*：是一个函数对象的“容器”，概念上像是函数指针类型的泛化，是一种“智能函数指针”，能够容纳任意符合函数签名的可调用对象，经常搭配 bind 使用。它的作者是 Doug Gregor。

function_types：提供了对函数、函数指针、函数引用和成员指针等类型进行分类、分解和合成的功能。它的作者是 Tobias Schwinger。

---

① 1.46 版之后中 filesystem 版本正式变更为 V3，有些类、接口发生变化（如取消了 basic_path，代之以统一的 path），但为了兼容仍然保留 V2。

functional：增强了 STL 中的函数对象适配器，并解决了“引用的引用”问题。它的作者是 Mark Rodgers。

functional/hash：实现了 TR1 中定义的散列函数，可以对 C++内置类型和标准库容器计算散列值，也可以拓展它以支持自定义类型。它的作者是 Daniel James。

functional/factory：工厂模式的实践，封装了 new 操作符。它的作者是 Tobias Schwinger。

functional/forward：函数对象的适配器，解决了使用右值的问题。它的作者是 Tobias Schwinger。

fusion：提供基于 tuple 的编译期容器和算法，是模板元编程的强大工具，可以与 mpl 很好的协同工作。它的作者是 Joel de Guzman、Dan Marsden 和 Tobias Schwinger。

G

geometry：几何图形处理库，功能包括求面积、周长、凸壳、距离等。它的作者是 Barend Gehrels、Bruno Lalande 和 Mateusz Loskot。

gil：是一个通用图像库，为像素、色彩、通道等图像处理概念提供了泛型的、STL 式的容器和算法，可以对图像做灰度化、梯度、均值、旋转等许多运算，支持 JPG、PNG、TIFF 等文件格式。它的作者是 Lubomir Bourdev 和 Hailin Jin。

graph：处理离散数学中的图结构，并提供图、矩阵等数据结构上的泛型算法，可以看做是 STL 在非线性容器领域的扩展。它的作者是 Jeremy Siek、Andrew Sutton 和 Jeremiah Willcock 等。

I

icl：提供了可以容纳“区间”的容器和其上的多种运算，不仅可以处理数学意义上的区间，也可以处理时间等其他领域的区间。它的作者是 Joachim Faulhaber。

integer*：提供了一组有关整数处理的头文件和类，具有良好的可移植性，让 C++能够更方便、更准确、更容易地处理整数类型。它的作者是 Beman Dawes 等。

interprocess：实现了可移植的进程间通信（IPC）的功能，并提供了简洁易用的 STL 风格接口。它的作者是 Ion Gaztanaga。

interval：处理“区间”概念相关的数学问题，把一般的算术运算扩展到了区间上，可以对区间执行各种运算。它的作者是 Guillaume Melquiond、HervBronnimann 和

Sylvain Pion。

intrusive：引入了日渐被遗忘的侵入式容器和算法，其接口类似于已被熟知的 STL，提供 list、tree、map 等与 STL 几乎等价的容器。它的作者是 Ion Gaztanaga。

in_place_factory：是工厂设计模式的一种实践，允许就地直接构造对象而不需要一个临时对象的拷贝。它的作者是 Fernando Cacciola。

io state savers*：可以简化恢复流状态的工作，保存流的当前状态，自动恢复流的状态或者由程序员控制恢复的时机。它的作者是 Daryle Walker。

iostreams：扩展了 C++标准库的流处理，建立了一个流处理框架，使得编写流式处理更加容易。它的作者是 Jonathan Turkanis。

iterators：定义了一组基于 STL 的新的迭代器概念、构造框架和有用的适配器，能够帮助程序员更轻松地实现迭代器模式。它的作者是 Dave Abrahams、Jeremy Siek 和 Thomas Witt。

L

lambda：为 C++引入了 lambda 表达式和函数式编程，可以就地创建小型的函数对象，避免函数对象的定义离调用点过远，方便代码维护。它的作者是 Jaakko Jarvi 和 Gary Powell。

lexical_cast*：进行“字面量”的转换，类似 C 中的 atoi 函数，可以进行字符串、整数/浮点数之间的字面转换。它的作者是 Kevlin Henney[①]。

M

math：包含了大量数学领域的模板类和算法。

math/complex number algorithms：复数反三角函数，它的作者是 John Maddock。

math/common_factor：最大公约数和最小公倍数，它的作者是 Daryle Walker。

math/octonion：八元数，它的作者是 Hubert Holin。

math/quaternion：四元数，它的作者是 Hubert Holin。

math/special_functions：大量近现代数学函数，它的作者是 John Maddock、

① 1.47 版中 lexical_cast 优化了性能，速度有提高。

Paul Bristow、Hubert Holin 和 Xiaogang Zhang。

math/statistical distributions：大量的统计分布和函数，它的作者是 John Maddock 和 Paul Bristow。

mem_fn：类似 bind 的函数绑定器，是标准库中 mem_fun/mem_fun_ref 的增强版本。它的作者是 Peter Dimov。

minmax*：对标准库中的算法 min/max 和 min_element/max_element 的增强，可在一次处理中同时获得最大最小值。它的作者是 Hervé Bronnimann。

MPI：用于高性能分布式并行计算应用的开发，封装了标准的 MPI（消息传送接口）以便更好地支持现代 C++编程风格。它的作者是 Douglas Gregor 和 Matthias Troyer。

mpl：模板元编程框架，包含有编译期的算法、容器和函数等完整的元编程工具。它的作者是 Aleksey Gurtovoy。

meta state machine：使用模板元编程技术实现的高性能的 UML2.0 有限状态机，速度快，结构良好。它的作者是 Christophe Henry。

multi_array*：是一个多维容器，高效地实现了 STL 风格的多维数组，比使用原始多维数组或者 vector<vector>更好。它的作者是 Ron Garcia。

multi_index：实现了具有多个 STL 兼容访问接口（索引）的容器。它的作者是 Joaquín M López Munoz。

N

numeric/conversion：提供用于安全数字转型的一组工具。它的作者是 Fernando Cacciola。

O

operators*：允许用户在自己的类里仅定义少量的操作符（如 operator<），就可方便地自动生成其他操作符重载，而且保证正确的语义实现。它的作者是 Dave Abrahams 和 Jeremy Siek。

optional*：使用“容器”语义，包装了“可能产生无效值”的对象，实现了“未初始化”的概念。它的作者是 Fernando Cacciola。

P

parameter：提供了类似 Python 和 c#的命名参数的特性（是在 C++标准化过程中曾经被拒绝的特性之一），可以使用参数名来指定函数参数值的机制，在实现有多个入口参数的函数时可以大大简化客户代码的编写，使用起来也更加方便。它的作者是 David Abrahams 和 Daniel Wallin。

phoenix：另一个函数式编程的强大工具，类似 lambda，可以就地定义匿名函数对象。它的作者是 Joel de Guzma、Dan Marsden 和 Thomas Heller。

pointer container：提供了与 STL 类似的若干种指针容器，包括 ptr_vector、ptr_list、ptr_map 等，性能较好且异常安全。它的作者是 Thorsten Ottosen。

polygon：这是由 Intel 赞助开发的一个功能强大的处理平面多边形的库。它的作者是 Lucanus Simonson。

pool*：基于简单分隔存储思想实现了一个快速、紧凑的内存池库，不仅能够管理大量的对象，还可以被用做 STL 的内存分配器。它的作者是 Steve Cleary。

preprocessor：提供类似于模板元编程的预处理元编程工具，发生在编译之前的预处理阶段。它的作者是 Vesa Karvonen 和 Paul Mensonides。

program_options*：提供了功能强大的命令行参数处理功能，不仅能够分析命令行，也能够从配置文件甚至环境变量中获取参数，实现了非常完善的程序配置选项处理功能。它的作者是 Vladimir Prus。

property map：是一个概念库，提供了 key-value 映射的属性概念定义，为从键到值的映射定义了一个通用接口。它的作者是 Jeremy Siek。

property tree*：是一个保存了多个属性值的树形数据结构，可以用类似路径的简单方式访问任意节点的属性，而且每个节点都可以用类似 STL 的风格遍历子节点。它的作者是 Marcin Kalicinski 和 Sebastian Redl。

proto：允许在 C++中构建专用领域嵌入式语言，基于表达式模板技术定义小型专用语言的“编译器”。它的作者是 Eric Niebler。

python*：可以方便和容易地在 Python 和 C++之间自由转换，全面支持 C++和 Python 的各种特性，包括 C++到 Python 的异常转换、默认参数、关键字参数、引用和指针等等，让 C++与 Python 可以近乎完美地对接。它的作者是 Dave Abrahams。

R

random*：可以产生高质量的随机数，并提供随机数发生器、分布等很多有用的数学、统计学相关概念。它的作者是 Jens Maurer[①]。

range：基于 STL 迭代器提出了“范围”的概念——容器的半开区间，相当于一个迭代器的 pair，有利于库作者编写新型算法。它的作者是 Thorsten Ottosen。

ratio：编译期的有理数运算，使用了模板元编程技术。它的作者是 Howard Hinnant、Beman Dewes 和 Vicente J. Botet Escribá。

rational*：实现了有理数概念，完善了 C++的数学域，运算时没有精度的损失。它的作者是 Paul Moore。

ref*：应用代理模式，引入对象引用的包装器概念解决了拷贝引用的问题，是一个“智能引用”。它的作者是 Jaako Jarvi、Peter Dimov、Doug Gregor 和 Dave Abrahams。

regex：正则表达式库，可用于字符串匹配、查找和替换。它的作者是 John Maddock。

result_of*：可以帮助程序员确定一个调用表达式的返回类型，主要用于泛型编程和其他 Boost 库组件。它的作者是 Doug Gregor[②]。

S

scope_exit：使用 preprocessor 库的预处理技术实现在退出作用域时的资源自动释放，也可以执行任意的代码。它的作者是 Alexander Nasonov。

serialization：实现了 C++数据结构的持久化，可以把任意的 C++对象整编为一串二进制字节流或者文本，然后再在需要的时候解整编恢复为原来的对象。它的作者是 Robert Ramey。

signals：实现了信号/插槽机制，是观察者模式的具体实践。它的作者是 Doug Gregor。

signals2*：基于 signals 库实现了线程安全的信号/插槽机制，而且无须编译即可使用。它的作者是 Frank Mori Hess。

smart_ptr*：是对 C++98 标准的一个绝佳补充，提供了六种智能指针，包括最“智能”

① 1.47 版中 random 增加了许多新的分布并修改了一些类的名字，此外还有一些较大的修改以便与 C++11 标准一致。

② 1.44 版增加了一个新的元函数 tr1_result_of<>。

的 shared_ptr。它的作者是 Greg Colvin、Beman Dawes、Peter Dimov 和 Darin Adler。

statechart：提供了一个功能完善的有限状态自动机框架，完全支持 UML 语义，可以从 UML 模型很方便地转换成 C++代码。它的作者是 Andreas Huber。

static_assert*：把断言的诊断时刻由运行期提前到编译期，让编译器检查可能发生的错误，能够更好地增加程序的健壮性。它的作者是 John Maddock。

spirit：是一个面向对象的递归下降解析器的生成框架，它使用 EBNF 语法，是一个比正则表达式更强大的语法分析器。它的作者是 Joel de Guzman 等。

string_algo*：是一个非常全面的字符串算法库，提供了大量的字符串操作函数，可以在不使用正则表达式的情况下处理大多数字符串相关问题。它的作者是 Pavol Droba。

swap*：是对标准库提供的 std::swap 的增强和泛化，为交换两个变量值提供了便捷的方法。它的作者是 Joseph Gauterin。

system*：使用轻量级的对象封装了操作系统底层的错误代码和错误信息，使调用操作系统功能的程序可以被很容易地移植到其他操作系统。它的作者是 Beman Dawes①。

T

test*：提供用于单元测试的基于命令行界面的测试套件，还附带有检测内存泄漏的功能，比其他的单元测试库更强大更方便好用。它的作者是 Gennadiy Rozental。

thread*：为 C++增加了线程处理的能力，提供简明清晰的线程、互斥量等概念，可以很容易地创建多线程应用程序，也是高度可移植的。它的作者是 William Kempf。

timer*：提供简易的度量时间和进度显示功能，可以用于性能测试等需要计时的任务，对于大多数的情况它足够用。它的作者是 Beman Dawes。

tokenizer*：是一个专门用于分词（token）的字符串处理库，可以使用简单易用的方法把一个字符串分解成若干单词。它的作者是 John Bandela。

TR1：是对 C++库扩展技术报告 TR1（Technical Report 1）的一个实现。但实际上它并没有真正地实现 TR1，而是对其他 Boost 库进行了包装，再导入到 std::tr1 名字空间中。它的作者是 John Maddock。

tribool*：类似 C++内置的 bool 类型，但基于三态的布尔逻辑。它的作者是 Doug

① 1.44 版开始 system_category 和 generic_category 由全局变量变为自由函数，因此可能需要修改原有代码。

Gregor。

tuple*：定义了一个有固定数目元素的容器，其中的每个元素类型都可以不相同，是 std::pair 的泛化，可以从函数返回任意数量的值，也可以代替 struct 组合数据。它的作者是 Jaakko Jarvi。

type_traits：提供一组特征（trait）类，用于在编译期确定类型是否具有某些特征。它的作者是 John Maddock、Steve Cleary 等。

typeof*：使用宏模拟了 C++11 新增加的 decltype 和 auto 关键字，可以减轻书写烦琐的变量类型声明的工作，简化代码。它的作者是 Arkadiy Vertleyb 和 Peder Holt。

U

uBLAS：是一个用于线性代数领域的数学库，比 std::valarray 要好得多，支持单位向量、稀疏向量、密集矩阵、稀疏矩阵、三角矩阵等许多线性代数概念。它的作者是 Joerg Walter 和 Mathias Koch。

units：实现了物理学的量纲处理，处理都是在编译期，没有运行时开销。它的作者是 Matthias Schabel 和 Steven Watanabe。

unordered*：提供了一个完全符合 C++新标准草案（TR1）的散列容器实现，包括无序集合（set）和无序映射（map）。它的作者是 Daniel James。

utility*：集合了很多非常有用的小工具，如 noncopyable、checked_delete 等。它的作者是 Dave Abrahams 等。

uuid*：是一个小的实用工具，可以表示和生成 UUID。它的作者是 Andy Tompkins[①]。

V

value_initialized：可以保证变量在声明时被正确地初始化，拥有零值或者缺省值。它的作者是 Fernando Cacciola。

variant*：与 any 有些类似，是一种可变类型，是对 C/C++中 union 概念的增强和扩展。它的作者是 Eric Friedman 和 Itay Maman。

① 1.44 版增加了自由函数 to_string()和 to_wstring()，可以直接把 uuid 转换为字符串，而且速度要比用 lexical_cast 更快。

W

wave：是使用 spirit 库开发的一个完全符合 C/C++标准的预处理器。它的作者是 Hartmut Kaiser。

X

xpressive*：一个先进的、灵活的、功能强大的正则表达式库，提供了对正则表达式的全面支持，甚至可以用来构建语法解析器，有动态和静态两种用法。它的作者是 Eric Niebler。

附录 C

程序员的工具箱

限于笔者自身条件，这里仅列出 Windows 和 Mac OS X 系统下的一些常用工具，供读者参考。

压缩解压缩

Windows 上的压缩工具首推 WinRAR，经典易用，自身的 RAR 格式速度快且压缩率高，同时几乎可以解压缩已知的所有压缩包格式（包括令人怀念的、古老的 DOS 时代的 ARJ，可惜没有 AIN），甚至还支持光盘镜像文件（ISO），绝对是程序员第一位必备的工具。同类可替代的软件还有 WinZip 和 7Zip。

Mac 自带对 Zip 格式的支持，但对其他常见压缩格式就无能为力了。收费的压缩工具有很多（例如 BetterZip），但 Mac AppStore 上可以免费获得强大的解压缩工具 The unachiver，它同 WinRAR 一样支持所有的压缩格式，用起来很方便。

文本编辑器

个人认为最佳编辑器应属 UltraEdit（同时提供 Windows 版和 Mac 版），功能无比强大，支持打开无限大的文件，支持 FTP、支持许多编程语言的语法颜色显示，支持十六进制编辑，支持列模式……可以说是“一旦拥有，别无所求”。它唯一的缺点是收费，而且价格不菲。如果要求不是太高可以考虑使用免费的 EditPlus、Notepad++和 UliPad，也都相当好用。

Mac 上的编辑器可以用 BBEdit，是与 UltraEdit 类似的强大编辑器，同样也要收费，不过它有一个免费的功能简化版的 TextWrangler 可用。我们也可以使用 Smultron 和 Fraise，它们的优点是轻巧方便，而且有中文界面。

（此外，经常工作在 Linux/Unix 上的程序员还可能会使用 Vim 和 Emacs，相信不用

笔者多说了。）

资料阅读

网络上流行的文档资料大都是 PDF 格式的，所以在 Windows 系统上 Adobe 公司本家出品的 PDF 文件阅读器 AcrobatReader 当然是必不可少的了（能够拥有功能更丰富强大的 Acrobat 就更好了）。Mac OS X 自带 PDF 支持，如果需要更多的阅读功能也可以安装一个。

同类软件还可以选择 FoxitReader（轻便的 PDF 阅读器）和浩瀚阅读器（超星格式）。

UML 工具

UML 建模工具不少都是企业重量级的，比如 Rational Software Architecture、PowerDesigner 和 Enterprise Architecture 等，功能虽强但使用时不免有“大炮打蚊子”的感觉。StarUML 是一个韩国人用 Delphi 开发的免费的 UML 建模工具，界面、操作非常类似经典的 RationalRose，小巧且强大，用起来非常方便，它还支持插件机制，可以扩展功能。遗憾的是 StarUML 自从 2008 年的 5.0 版本后就没有更新了。

StarUML 只有 Windows 版本，在 Mac 上我们可以使用免费的 ArgoUML，功能类似。

思维导图

思维导图又称心智图，工作和会议中用来整理思路、发散性思维都很不错，Windows 上较好的有 MindManager，Mac 上也有 MindNote，而 FreeMind 则是跨平台的，这些软件都是免费的。

虚拟机

程序员经常要做跨平台开发，虚拟机这种工具大大降低了我们切换平台的成本。VirtualBox 是 Oracle（原 Sun）公司出品的完全免费的虚拟机软件，可以在宿主机上安装多个其他的操作系统，兼容性和效率都不错。另外一个不能不提的就是虚拟机的“老大”VMWare 了，历史悠久功能强大使用方便，不过价格不菲。

此外，在 Linux 系统上还可以选择开源的 Xen，以及专门虚拟 Windows 系统的 Wine 和 Crossover。

开发辅助

“VC 程序员都爱吃西红柿”——由 WholeTomato 公司开发的 VC 插件 VisualAssistX 差不多是每一个 VC 程序员都必备的工具，它为 VC 略显简陋的开发环境增加了很多有用的功能，例如.h/.cpp 切换、前进后退导航、语法色彩显示等，还有现代集成开发环境必备的自

动重构功能，使用它能够极大地提高 VC 程序员的工作效率。

SourceInsight 是另一个可以查看代码间相互关系的好软件，用它来分析阅读代码非常便利，可惜的是它只有 Windows 版。

文件管理

Total Commander（即原 Windows Commander）是 Windows 资源管理器的增强版本，使用双窗口可以更容易地进行各种文件操作，同时它也是一个很好的 FTP 工具，可以配合自己喜欢的文本编辑器调试开发后台程序。Mac OS X 下也有类似的增强工具 ForkLift、PathFinder 等可用。

Beyond Compare 是一个非常强大好用的比较/同步工具，不仅可以比较文本文件，也可以比较二进制文件和目录，用来备份或同步都很方便，可惜的是它没有 Mac 版。

光盘镜像工具

在 Windows 上作者常用的虚拟光盘软件是 Alcohol 120%和 UltraISO——都是用于处理光盘的超级武器，不仅可以创建或编辑 CD/DVD 镜像文件，还能够虚拟光驱、烧录光盘、拷贝光盘，支持市面上常见的许多光盘镜像格式[1]。

同类软件还有 WinISO、CloneCD 和适用于 Mac 的 Burn。

办公软件

办公套件首推微软出品的 Office，不用再多说了，里面的 Word、Excel 和 PowerPoint 三大件是每一个计算机用户都必须掌握的基本技能，几乎成为了业界的办公标准。如果支持国产软件可以选择 WPSOffice，其他的免费办公软件则有跨平台的 LibreOffice、NeoOffice。

Mac 上笔者通常使用 Apple 自家的 iWork 套件：Keynote、Pages 和 Numbers，它们分别对应 Office 的 PowerPoint、Word 和 Excel，不过 iWork 对 Office 格式支持的不够好，所以有时还是得使用 Mac 版的 Office。

电子词典

阅读英文资料必须要备着一个较好的电子词典，这里面最老牌的就是金山词霸了，后起之秀则是网易的有道词典。如果能够上 Internet，那就更方便了，可以直接用 Google 或者维基百科搜索更准确及时的词义。

① 题外话，随着大硬盘和宽带网络的流行，如今光盘/光驱的使用率越来越低了，新版的 Mac mini 甚至完全取消了内置光驱。